iBATIS.NET

DataMapper Developer Guide

Version 1.5.1

July 2006

Authors - Ted Husted, Gilles Bayon, Clinton Begin, Roberto Rabe

Table of Contents

Chapter 1. Introduction

1.1. Overview

The iBATIS DataMapper framework makes it easier to use a database with a Java or .NET application. iBATIS DataMapper couples objects with stored procedures or SQL statements using a XML descriptor. Simplicity is the biggest advantage of the iBATIS DataMapper over object relational mapping tools. To use iBATIS DataMapper you rely on your own objects, XML, and SQL. There is little to learn that you don't already know. With iBATIS DataMapper you have the full power of both SQL and stored procedures at your fingertips.

1.2. What's covered here

This Guide covers the .NET implementations of iBATIS DataMapper. The Java implementation offers the same services with some changes in the API.

Since iBATIS relies on an XML descriptor to create the mappings, much of the material applies to both implementations.

For installation instructions, see the section called the .NET Developer Guide.

A Tutorial is also available. We recommend reviewing the Tutorial for your platform before reading this Guide.

Tip

If you would like to get the latest development (unreleased) version of this Guide, please see the iBATIS Wiki FAQ [http://opensource.atlassian.com/confluence/oss/display/IBATIS/]. The FAQ entry explains how you can access our SVN source repository and generate CHM and PDF files of the latest development documentation for the DataMapper.

1.3. License Information

1.4. Support

Mailing lists and bug trackers are available (courtesy of Apache Software Foundation) at iBATIS's Apache project page. Just direct your browser to http://ibatis.apache.org/.

1.5. Disclaimer

INFORMATION IN THIS DOCUMENT. The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Chapter 2. The Big Picture

2.1. Introduction

iBATIS is a simple but complete framework that makes it easy for you to map your objects to your SQL statements or stored procedures. The goal of the iBATIS framework is to obtain 80% of data access functionality using only 20% of the code.

2.2. What does it do?

Developers often create maps between objects within an application. One definition of a Mapper is an "object that sets up communication between two independent objects." A Data Mapper is a "layer of mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself. " [Patterns of Enterprise Architecture, ISBN 0-321-12742-0].

You provide the database and the objects; iBATIS provides the mapping layer that goes between the two.

2.3. How does it work?

Your programming platform already provides a capable library for accessing databases, whether through SQL statements or stored procedures. But developers find several things are still hard to do well when using "stock" ADO.NET, including:

- Separating SQL code from programming code
- Passing input parameters to the library classes and extracting the output
- Separating data access classes from business logic classes
- Caching often-used data until it changes
- Managing transactions and threading

iBATIS DataMapper solves these problems -- and many more -- by using XML documents to create a mapping between a plain-old object and a SQL statement or a stored procedure. The "plain-old object" can be a IDictionary or property object.

Tip

The object does not need to be part of a special object hierarchy or implement a special interface. (Which is why we call them "plain-old" objects.) Whatever you are already using should work just fine.

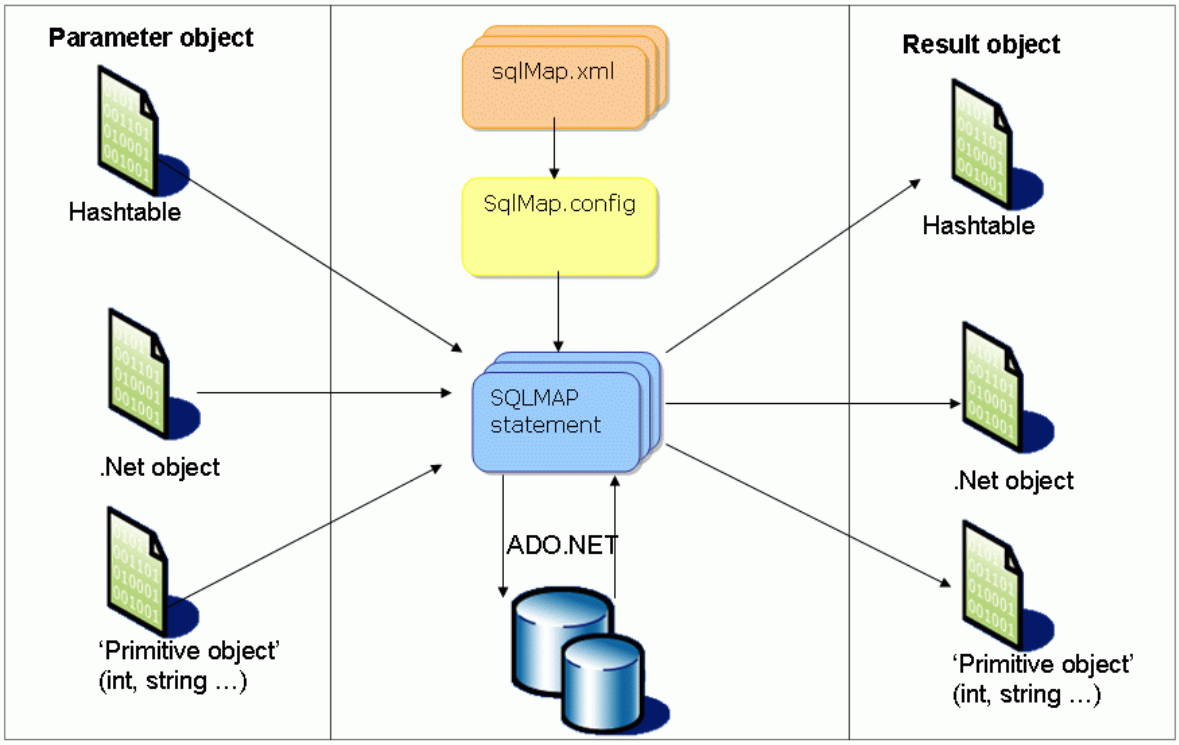

Figure 2.1. iBATIS DataMapper workflow

Here's a high level description of the workflow diagrammed by Figure 2.1:

1. Provide a parameter, either as an object or a native type. The parameter can be used to set runtime values in your SQL statement or stored procedure. If a runtime value is not needed, the parameter can be omitted.

2. Execute the mapping by passing the parameter and the name you gave the statement or procedure in your XML descriptor. This step is where the magic happens. The framework will prepare the SQL statement or stored procedure, set any runtime values using your parameter, execute the procedure or statement, and return the result.

3. In the case of an update, the number of rows affected is returned. In the case of a query, a single object, or a collection of objects is returned. Like the parameter, the result object, or collection of objects, can be a plain-old object or a native type.

So, what does all this look like in your source code? Here's how you might code the insert of a "lineItem" object into your database:

```
C#
Mapper.Instance().Insert("InsertLineItem",lineItem);
```

If your database is generating the primary keys, the generated key can be returned from the same method call, like this:

```
C#
int myKey = Mapper.Instance().Insert("InsertLineItem",lineItem);
```

Example 2.1 shows an XML descriptor for "InsertLineItem".

Example 2.1. The "InsertLineItem" descriptor

```
<insert id="InsertLineItem" parameterClass="LineItem">
  INSERT INTO [LinesItem]
    (Order_Id, LineItem_LineNum, Item_Id, LineItem_Quantity, LineItem_UnitPrice)
  VALUES
    (#Order.Id#, #LineNumber#, #Item.Id#, #Quantity#, #Item.ListPrice#)
 <selectKey type="post" resultClass="int" property="Id" >
  select @@IDENTITY as value
 </selectKey>
</insert>
```

The <selectKey> stanza returns an autogenerated key from a SQL Server database (for example).

If you need to select multiple rows, iBATIS can return a list of objects, each mapped to a row in the result set:

```
C#
IList productList = Mapper.Instance().QueryForList("selectProduct",categoryKey);
```

Or just one, if that's all you need:

```
C#
Product product = Mapper.Instance().QueryForObject("SelectProduct",productKey) as Product;
```

Of course, there's more, but this is iBATIS from 10,000 meters. (For a longer, gentler introduction, see the Tutorial.) Section 3 describes the Data Map definition files -- where the statement for "InsertLineItem" would be defined. The Developers Guide for your platform (Section 4) describes the "bootstrap" configuration file that exposes iBATIS to your application.

2.4. Is iBATIS the best choice for my project?

iBATIS is a Data Mapping tool. Its role is to map the columns of a database query (including a stored procedure) to the properties of an object. If your application is based on business objects (including Maps or IDictionary objects), then iBATIS can be a good choice. iBATIS is an even better choice when your application is layered, so that that the business layer is distinct from the user-interface layer.

Under these circumstances, another good choice would be an Object/Relational Mapping tool (OR/M tool), like NHibernate. Other products in this category are Apache ObjectRelationalBridge and Gentle.NET. An OR/M tool generates all or most of the SQL for you, either beforehand or at runtime. These products are called OR/M tools because they try to map an object graph to a relational schema.

iBATIS is not an OR/M tool. iBATIS helps you map objects to stored procedures or SQL statements. The underlying schema is irrelevant. An OR/M tool is great if you can map your objects to tables. But they are not so great if your objects are stored as a relational view rather than as a table. If you can write a statement or procedure that exposes the columns for your object, regardless of how they are stored, iBATIS can do the rest.

So, how do you decide whether to OR/M or to DataMap? As always, the best advice is to implement a representative part of your project using either approach, and then decide. But, in general, OR/M is

a good thing when you

1. Have complete control over your database implementation
2. Do not have a Database Administrator or SQL guru on the team
3. Need to model the problem domain outside the database as an object graph.

Likewise, the best time to use a Data Mapper, like iBATIS, is when:

1. You do *not* have complete control over the database implementation, or want to continue to access a legacy database as it is being refactored.
2. You have database administrators or SQL gurus on the team.
3. The database is being used to model the problem domain, and the application's primary role is to help the client use the database model.

In the end, *you* have to decide what's best for *your* project. If a OR/M tool works better for you, that's great! If your next project has different needs, then we hope you give iBATIS another look. If iBATIS works for you now: Excellent! Join the iBATIS user mailing list if you have any questions.

Chapter 3. Working with Data Maps

3.1. Introduction

If you want to know how to configure and install iBATIS, see the Developer Guide section for your platform. But if you want to know how iBATIS *really* works, continue from here.

The Data Map definition file is where the interesting stuff happens. Here, you define how your application interacts with your database. As mentioned, the Data Map definition is an XML descriptor file. By using a service routine provided by iBATIS, the XML descriptors are rendered into a client object (or `Mapper`). To access your Data Maps, your application calls the client object and passes in the name of the statement you need.

The real work of using iBATIS is not so much in the application code, but in the XML descriptors that iBATIS renders. Instead of monkeying with application source code, you monkey with XML descriptors instead. The benefit is that the XML descriptors are much better suited to the task of mapping your object properties to database entities. At least, that's our own experience with our own applications. Of course, your mileage may vary.

3.2. What's in a Data Map definition file, anyway?

If you read the Tutorial, you've already seen some simple Data Map examples, like the one shown in Example 2.1.

Example 3.1. A simple Data Map (.NET)

```
<?xml version="1.0" encoding="UTF-8" ?>
  <sqlMap namespace="LineItem"
xmlns="http://ibatis.apache.org/mapping"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >

  <!--Type aliases allow you to use a shorter name for long fully qualified class names.-->
  <alias>
    <typeAlias alias="LineItem" type="NPetshop.Domain.Billing.LineItem, NPetshop.Domain" />
  </alias>

  <statements>
    <insert id="InsertLineItem" parameterClass="LineItem">
      INSERT INTO [LinesItem]
        (Order_Id, LineItem_LineNum, Item_Id, LineItem_Quantity, LineItem_UnitPrice)
      VALUES
       (#Order.Id#, #LineNumber#, #Item.Id#, #Quantity#, #Item.ListPrice#)
    </insert>
  </statements>
</sqlMap>
```

This map takes some properties from a LineItem instance and merges the values into the SQL statement. The value-add is that our SQL in separated from our program code, and we can pass our LineItem instance directly to a library method:

```
C#
Mapper.Instance().Insert("InsertLineItem",lineItem);
```

No fuss, no muss. Likewise, see Example 3.2 for a simple select statement.

In Example 3.1, we use SQL aliasing to map columns to our object properties and an iBATIS inline parameter (see sidebar) to insert a runtime value. Easy as pie.

A Quick Glance at Inline Parameters

Say we have a mapped statement element that looks like this:

```
<statement id="InsertProduct">
  insert into Products (Product_Id, Product_Description)
  values (#Id#, #Description#);
</statement>
```

The inline parameters here are #Id# and #Description#. Let's also say that we have an object with the properties *`Id`* and *`Description`*. If we set the object properties to *`5`* and "*`dog`*", respectively, and passed the object to the mapped statement, we'd end up with a runtime query that looked like this:

```
insert into Products (Product_Id, Product_Description) values (5, 'dog');
```

For more about inline parameters, see Section 3.4.

But, what if you wanted some ice cream with that pie? And maybe a cherry on top? What if we wanted to cache the result of the select? Or, what if we didn't want to use SQL aliasing or named parameters. (Say, because we were using pre-existing SQL that we didn't want to touch.) Example 3.2 shows a Data Map that specifies a cache, and uses a <parameterMap> and a <resultMap> to keep our SQL pristine.

Example 3.2. A Data Map definition file with some bells and whistles

```
<?xml version="1.0" encoding="UTF-8" ?>
  <sqlMap namespace="Product"
xmlns="http://ibatis.apache.org/mapping"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >

  <alias>
    <typeAlias alias="Product" type="Example.Domain.Product, Example.Domain" />
  </alias>

  <cacheModels>
    <cacheModel id="productCache" implementation="LRU">
      <flushInterval hours="24"/>
      <property name="CacheSize" value="1000" />
    </cacheModel>
  </cacheModels>

   <resultMaps>
    <resultMap id="productResult" class="Product">
      <result property="Id" column="Product_Id"/>
      <result property="Description" column="Product_Description"/>
    </resultMap>
   </resultMaps>

  <statements>
    <select id="GetProduct" parameterMap="productParam" cacheModel="productCache">
      select * from Products where Product_Id = ?
    </select>
  </statements>
```

```
  <parameterMaps>
    <parameterMap id="productParam" class="Product">
      <parameter property="Id"/>
    </parameterMap>
  <parameterMaps>

</sqlMap>
```

In Example 3.2, <parameterMap> maps the SQL "?" to the `product` Id property. The <resultMap> maps the columns to our object properties. The <cacheModel> keeps the result of the last one thousand of these queries in active memory for up to 24 hours.

Example 3.2 is longer and more complex than Example 3.1, but considering what you get in return, it seems like a fair trade. (A bargain even.)

Many *agile* developers would start with something like Example 3.1 and add features like caching later. If you changed the Data Map from Example 3.1 to Example 3.2, you would not have to touch your application source code at all. You can start simple and add complexity only when it is needed.

A single Data Map definition file can contain as many Cache Models, Type Aliases, Result Maps, Parameter Maps, and Mapped Statements (including stored procedures), as you like. Everything is loaded into the same configuration, so you can define elements in one Data Map and then use them in another. Use discretion and organize the statements and maps appropriately for your application by finding some logical way to group them.

3.3. Mapped Statements

Mapped Statements can hold any SQL statement and can use Parameter Maps and Result Maps for input and output. (A stored procedure is a specialized form of a statement. See section 3.3.1 and 3.3.2 for more information.)

If the case is simple, the Mapped Statement can reference the parameter and result classes directly. Mapped Statements support caching through reference to a Cache Model element. The following example shows the syntax for a statement element.

Example 3.3. Statement element syntax

```
<statement id="statement.name"
  [parameterMap="parameterMap.name"]
  [parameterClass="alias"]
  [resultMap="resultMap.name"]
  [resultClass="class.name|alias"]
  [listClass="class.name|alias"]
  [cacheModel="cache.name"]
  [extends="statement.name"]
>

  select * from Products where Product_Id = [?|#propertyName#]
  order by [$simpleDynamic$]

</statement>
```

In Example 3.3, the [bracketed] parts are optional, and some options are mutually exclusive. It is perfectly legal to have a Mapped Statement as simple as shown by Example 3.4.

Example 3.4. A simplistic Mapped Statement

```
<statement id="InsertTestProduct" >
  insert into Products (Product_Id, Product_Description) values (1, "Shih Tzu")
</statement>
```

Example 3.4 is obviously unlikely, unless you are running a test. But it does shows that you can use iBATIS to execute arbitrary SQL statements. More likely, you will use the object mapping features with Parameter Maps (Section 3.4) and Result Maps (Section 3.5) since that's where the magic happens.

3.3.1. Statement Types

The <statement> element is a general "catch all" element that can be used for any type of SQL statement. Generally it is a good idea to use one of the more specific statement-type elements. The more specific elements provided better error-checking and even more functionality. (For example, the insert statement can return a database-generated key.) Table 3.1 summarizes the statement-type elements and their supported attributes and features.

Table 3.1. The six statement-type elements

Statement Element	Attributes	Child Elements	Methods
<statement>	id parameterClass resultClass listClass parameterMap resultMap cacheModel	All dynamic elements	Insert Update Delete All query methods
<insert>	id parameterClass parameterMap	All dynamic elements <selectKey> <generate>	Insert Update Delete
<update>	id parameterClass parameterMap extends	All dynamic elements <generate>	Insert Update Delete
<delete>	id parameterClass parameterMap extends	All dynamic elements <generate>	Insert Update Delete
<select>	id parameterClass resultClass listClass parameterMap resultMap cacheModel	All dynamic elements <generate>	All query methods

Statement Element	Attributes	Child Elements	Methods
	`extends`		

<procedure>	`id`
	`parameterMap`
	`resultClass`
	`resultMap`
	`cacheModel`
		`All dynamic elements`	`Insert`
			`Update`
			`Delete`
			`All query methods`

The various attributes used by statement-type elements are covered in Section 3.3.4.

3.3.2. Stored Procedures

iBATIS DataMapper treats stored procedures as yet another statement type. Example 3.5 shows a simple Data Map hosting a stored procedure.

Example 3.5. A Data Map using a stored procedure

```
<!-- Microsot SQL Server -->
<procedure id="SwapEmailAddresses" parameterMap="swap-params">
  ps_swap_email_address
</procedure>
...
<parameterMap id="swap-params">
  <parameter property="email1" column="First_Email" />
  <parameter property="email2" column="Second_Email" />
</parameterMap>

<!-- Oracle with MS OracleClient provider -->
<procedure id="InsertCategory" parameterMap="insert-params">
 prc_InsertCategory
</procedure>
...
<parameterMap id="insert-params">
 <parameter property="Name"       column="p_Category_Name"/>
 <parameter property="GuidString" column="p_Category_Guid" dbType="VarChar"/>
 <parameter property="Id"         column="p_Category_Id"   dbType="Int32"   type="Int"/>
</parameterMap>

<!-- Oracle with ODP.NET 10g provider -->
<statement id="InsertAccount" parameterMap="insert-params">
 prc_InsertAccount
</statement>
...
<parameterMap id="insert-params">
 <parameter property="Id"           dbType="Int32"/>
 <parameter property="FirstName"    dbType="VarChar2" size="32"/>
 <parameter property="LastName"     dbType="VarChar2" size="32"/>
 <parameter property="EmailAddress" dbType="VarChar2" size="128"/>
</parameterMap>
```

The idea behind Example 3.5 is that calling the stored procedure SwapEmailAddresses would exchange two email addresses between two columns in a database table and also in the parameter object (a `HashTable`). The parameter object is only modified if the parameter mappings mode attribute is set to *`InputOutput`* or *`Output`*. Otherwise they are left unchanged. Of course, immutable parameter objects (e.g. `String`) cannot be modified.

Note

For .NET, the parameterMap attribute is required. The DBType, parameter direction, size, precision,

and scale are usually auto-discovered by the framework (via the CommandBuilder) depending on your provider.

3.3.3. The SQL

If you are not using stored procedures, the most important part of a statement-type element is the SQL. You can use any SQL statement that is valid for your database system. Since iBATIS passes the SQL through to the standard libraries (ADO.NET), you can use any statement with iBATIS that you could use without iBATIS. You can use whatever functions your database system supports, and even send multiple statements, so long as your driver or provider supports them.

If standard, static SQL isn't enough, iBATIS can help you build a dynamic SQL statement. See Section 3.9 for more about Dynamic SQL.

3.3.3.1. Escaping XML symbols

Because you are combining SQL and XML in a single document, conflicts can occur. The most common conflict is the greater-than and less-than symbols (><). SQL statements use these symbols as operators, but they are reserved symbols in XML. A simple solution is to *escape* the SQL statements that uses XML reserved symbols within a CDATA element. Example 3.6 demonstrates this.

Example 3.6. Using CDATA to "escape" SQL code

```
<statement id="SelectPersonsByAge" parameterClass="int" resultClass="Person">
  <![CDATA[
     SELECT * FROM PERSON WHERE AGE > #value#
  ]]>
</statement>
```

3.3.3.2. Auto-Generated Keys

Many database systems support auto-generation of primary key fields, as a vendor extension. Some vendors pre-generate keys (e.g. Oracle), some vendors post-generate keys (e.g. MS-SQL Server and MySQL). In either case, you can obtain a pre-generated key using a <selectKey> stanza within an <insert> element. Example 3.7 shows an <insert> statement for either approach.

Example 3.7. <insert> statements using <selectKey> stanzas

```
<!—Oracle SEQUENCE Example using .NET 1.1 System.Data.OracleClient -->
<insert id="insertProduct-ORACLE" parameterClass="product">
  <selectKey resultClass="int" type="pre" property="Id" >
     SELECT STOCKIDSEQUENCE.NEXTVAL AS VALUE FROM DUAL
  </selectKey>
  insert into PRODUCT (PRD_ID,PRD_DESCRIPTION) values (#id#,#description#)
</insert>

<!— Microsoft SQL Server IDENTITY Column Example -->
<insert id="insertProduct-MS-SQL" parameterClass="product">
  insert into PRODUCT (PRD_DESCRIPTION)
  values (#description#)
 <selectKey resultClass="int" type="post" property="id" >
   select @@IDENTITY as value
 </selectKey>
</insert>
```

```
<!-- MySQL Example -->
<insert id="insertProduct-MYSQL" parameterClass="product">
  insert into PRODUCT (PRD_DESCRIPTION)
  values (#description#)
 <selectKey resultClass="int" type="post" property="id" >
   select LAST_INSERT_ID() as value
 </selectKey>
</insert>
```

3.3.3.3. <generate> tag

You can use iBATIS to execute any SQL statement your application requires. When the requirements for a statement are simple and obvious, you may not even need to write a SQL statement at all. The <generate> tag can be used to create simple SQL statements automatically, based on a <parameterMap> element. The four CRUD statement types (insert, select, update, and delete) are supported. For a select, you can select all or select by a key (or keys). Example 3.8 shows an example of generating the usual array of CRUD statements.

Example 3.8. Creating the "usual suspects" with the <generate> tag

```
<parameterMaps>
  <parameterMap id="insert-generate-params">
    <parameter property="Name" column="Category_Name"/>
    <parameter property="Guid" column="Category_Guid" dbType="UniqueIdentifier"/>
  </parameterMap>

  <parameterMap id="update-generate-params" extends="insert-generate-params">
    <parameter property="Id" column="Category_Id" />
  </parameterMap>

  <parameterMap id="delete-generate-params">
    <parameter property="Id" column="Category_Id" />
    <parameter property="Name" column="Category_Name"/>
  </parameterMap>

  <parameterMap id="select-generate-params">
    <parameter property="Id" column="Category_Id" />
    <parameter property="Name" column="Category_Name"/>
    <parameter property="Guid" column="Category_Guid" dbType="UniqueIdentifier"/>
  </parameterMap>

</parameterMaps>

<statements>

  <update id="UpdateCategoryGenerate" parameterMap="update-generate-params">
    <generate table="Categories" by="Category_Id"/>
  </update>

  <delete id="DeleteCategoryGenerate" parameterMap="delete-generate-params">
    <generate table="Categories" by="Category_Id, Category_Name"/>
  </delete>

  <select id="SelectByPKCategoryGenerate" resultClass="Category" parameterClass="Category"
          parameterMap="select-generate-params">
    <generate table="Categories" by="Category_Id"/>
  </select>

  <select id="SelectAllCategoryGenerate" resultClass="Category"
          parameterMap="select-generate-params">
    <generate table="Categories" />
  </select>

  <insert id="InsertCategoryGenerate" parameterMap="insert-generate-params">
    <selectKey property="Id" type="post" resultClass="int">
      select @@IDENTITY as value
    </selectKey>
    <generate table="Categories" />
```

```
  </insert>

</statements>
```

Note

The SQL is generated when the DataMapper instance is built, so there is no performance impact at execution time.
The tag generates ANSI SQL, which should work with any compliant database. Special types, such as blobs, are not supported, and vendor-specific types are also not supported. But, the generate tag does keep the simple things simple.

Important

The intended use of the <generate> tag is to save developers the trouble of coding mundane SQL statements (and only mundane statements). It is not meant as a object-to-relational mapping tool. There are many frameworks that provide extensive object-to-relational mapping features. The <generate> tag is not a replacement for any of those. When the <generate> tag does not suit your needs, use a conventional statement instead.

3.3.3.3.1. <generate> tag attributes

The generate tag supports two attributes :

Table 3.2. <generate> attributes

Attribute	Description	Required
table	specifies the table name to use in the SQL statement.	yes
by	specifies the columns to use in a WHERE clause	no

3.3.4. Statement-type Element Attributes

The six statement-type elements take various attributes. See Section 3.3.1 for a table itemizing which attributes each element-type accepts. The individual attributes are described in the sections that follow.

3.3.4.1. id

The required *id* attribute provides a name for this statement, which must be unique within this <SqlMap>.

3.3.4.2. parameterMap

A Parameter Map defines an ordered list of values that match up with the "*?*" placeholders of a standard, parameterized query statement. Example 3.9 shows a <parameterMap> and a corresponding <statement>.

Example 3.9. A parameterMap and corresponding statement

```
<parameterMap id="insert-product-param" class="Product">
  <parameter property="id"/>
  <parameter property="description"/>
</parameterMap>

<statement id="insertProduct" parameterMap="insert-product-param">
  insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (?,?);
</statement>
```

In Example 3.9, the Parameter Map describes two parameters that will match, in order, two placeholders in the SQL statement. The first "*?*" is replaced by the value of the id property. The second is replaced with the description property.

iBATIS also supports named, inline parameters, which most developers seem to prefer. However, Parameter Maps are useful when the SQL must be kept in a standard form or when extra information needs to be provided. For more about Parameter Maps see Section 3.4.

3.3.4.3. parameterClass

If a parameterMap attribute is not specified, you may specify a parameterClass instead and use inline parameters (see Section 3.4.3). The value of the parameterClass attribute can be a Type Alias or the fully qualified name of a class. Example 3.10 shows a statement using a fully-qualified name versus an alias.

Example 3.10. Ways to specify a parameterClass

```
<!-- fully qualified classname -->
<statement id="statementName" parameterClass="Examples.Domain.Product, Examples.Domain">
  insert into PRODUCT values (#id#, #description#, #price#)
</statement>

<!-- typeAlias (defined elsewhere) -->
<statement id="statementName" parameterClass="Product">
  insert into PRODUCT values (#id#, #description#, #price#)
</statement>
```

3.3.4.4. resultMap

A Result Map lets you control how data is extracted from the result of a query, and how the columns are mapped to object properties. Example 3.11 shows a <resultMap> element and a corresponding <statement> element.

Example 3.11. A <resultMap> and corresponding <statement>

```
<resultMap id="select-product-result" class="product">
  <result property="id" column="PRD_ID"/>
  <result property="description" column="PRD_DESCRIPTION"/>
</resultMap>

<statement id="selectProduct" resultMap="select-product-result">
  select * from PRODUCT
```

```
</statement>
```

In Example 3.11, the result of the SQL query will be mapped to an instance of the Product class using the "select-product-result" <resultMap>. The <resultMap> says to populate the id property from the PRD_ID column, and to populate the *description* property from the PRD_DESCRIPTION column.

Tip

In Example 3.11, note that using " select * " is supported. If you want all the columns, you don't need to map them all individually. (Though many developers consider it a good practice to always specify the columns expected.)

For more about Result Maps, see Section 3.5.

3.3.4.5. resultClass

If a resultMap is not specified, you may specify a resultClass instead. The value of the resultClass attribute can be a Type Alias or the fully qualified name of a class. The class specified will be automatically mapped to the columns in the result, based on the result metadata. The following example shows a <statement> element with a resultClass attribute.

Example 3.12. A <statement >element with resultClass attribute

```
<statement id="SelectPerson" parameterClass="int" resultClass="Person">
  SELECT
  PER_ID as Id,
  PER_FIRST_NAME as FirstName,
  PER_LAST_NAME as LastName,
  PER_BIRTH_DATE as BirthDate,
  PER_WEIGHT_KG as WeightInKilograms,
  PER_HEIGHT_M as HeightInMeters
  FROM PERSON
  WHERE PER_ID = #value#
</statement>
```

In Example 3.12, the `Person` class has properties including: Id, FirstName, LastName, BirthDate, WeightInKilograms, and HeightInMeters. Each of these corresponds with the column aliases described by the SQL select statement using the "as" keyword –a standard SQL feature. When executed, a `Person` object is instantiated and populated by matching the object property names to the (aliased) column names from the query.

Using SQL aliases to map columns to properties saves defining a <resultMap> element, but there are limitations. There is no way to specify the types of the output columns (if needed), there is no way to automatically load related data such as complex properties, and there is a slight performance consequence from accessing the result metadata. Architecturally, using aliases this way mixes *database logic* with *reporting logic*, making the query harder to read and maintain. You can overcome these limitations with an explicit Result Map (Section 3.5).

3.3.4.6. listClass

In addition to providing the ability to return an IList of objects, the DataMapper supports the use of a

strongly-typed custom collection: a class that implements the System.Collections.CollectionBase abstract class. The following is an example of a CollectionBase class that can be used with the DataMapper.

Example 3.13. A System.Collections.CollectionBase implementation

```
using System;
using System.Collections;

namespace WebShop.Domain
{
 public class AccountCollection : CollectionBase
 {
  public AccountCollection() {}

  public Account this[int index]
  {
   get { return (Account)List[index]; }
   set { List[index] = value; }
  }

  public int Add(Account value)
  {
   return List.Add(value);
  }

  public void AddRange(Account[] value)
  {
   for (int i = 0; i < value.Length; i++)
   {
    Add(value[i]);
   }
  }

  public void AddRange(AccountCollection value)
  {
   for (int i = 0; i < value.Count; i++)
   {
    Add(value[i]);
   }
  }

  public bool Contains(Account value)
  {
   return List.Contains(value);
  }

  public void CopyTo(Account[] array, int index)
  {
   List.CopyTo(array, index);
  }

  public int IndexOf(Account value)
  {
   return List.IndexOf(value);
  }

  public void Insert(int index, Account value)
  {
   Account.Insert(index, value);
  }

  public void Remove(Account value)
  {
   Account.Remove(value);
  }
 }
}
```

A CollectionBase class can be specified for a select statement through the listClass attribute. The value of the listClass attribute can be a Type Alias or the fully qualified name of a class. The statement should also indicate the resultClass so that the DataMapper knows how to handle the type of objects in the collection. The resultClass specified will be automatically mapped to the columns in the result, based on the result metadata. The following example shows a <statement> element with a listClass attribute.

Example 3.14. A <statement >element with listClass attribute

```
<statement id="GetAllAccounts"
 listClass="AccountCollection"
 resultClass="Account">
   select
   Account_ID as Id,
   Account_FirstName as FirstName,
   Account_LastName as LastName,
   Account_Email as EmailAddress
   from Accounts
   order by Account_LastName, Account_FirstName
</statement>
```

3.3.4.7. cacheModel

If you want to cache the result of a query, you can specify a Cache Model as part of the <statement> element. Example 3.15 shows a <cacheModel> element and a corresponding <statement>.

Example 3.15. A <cacheModel> element with its corresponding <statement>

```
<cacheModel id="product-cache" implementation="LRU">
  <flushInterval hours="24"/>
  <flushOnExecute statement="insertProduct"/>
  <flushOnExecute statement="updateProduct"/>
  <flushOnExecute statement="deleteProduct"/>
  <property name="size" value="1000" />
</cacheModel>

<statement id="selectProductList" parameterClass="int" cacheModel="product-cache">
  select * from PRODUCT where PRD_CAT_ID = #value#
</statement>
```

In Example 3.15, a cache is defined for products that uses a LRU reference type and flushes every 24 hours or whenever associated update statements are executed. For more about Cache Models, see Section 3.8.

3.3.4.8. extends

When writing Sql, you often encounter duplicate fragments of SQL. iBATIS offers a simple yet powerful attribute to reuse them.

```
<select id="GetAllAccounts"
  resultMap="indexed-account-result">
```

```
select
  Account_ID,
  Account_FirstName,
  Account_LastName,
  Account_Email
from Accounts
</select>

<select id="GetAllAccountsOrderByName"
  extends="GetAllAccounts"
  resultMap="indexed-account-result">
    order by Account_FirstName
</select>
```

3.4. Parameter Maps and Inline Parameters

Most SQL statements are useful because we can pass them values at runtime. Someone wants a database record with the ID 42, and we need to merge that ID number into a select statement. A list of one or more parameters are passed at runtime, and each placeholder is replaced in turn. This is simple but labor intensive, since developers spend a lot of time counting symbols to make sure everything is in sync.

Note

Preceding sections briefly touched on inline parameters, which automatically map properties to named parameters. Many iBATIS developers prefer this approach. But others prefer to stick to the standard, anonymous approach to SQL parameters by using parameter maps. Sometimes people need to retain the purity of the SQL statements; other times they need the detailed specification offered by parameter maps due to database or provider-specific information that needs to be used.

A Parameter Map defines an ordered list of values that match up with the placeholders of a parameterized query statement. While the attributes specified by the map still need to be in the correct order, each parameter is named. You can populate the underlying class in any order, and the Parameter Map ensures each value is passed in the correct order.

Note

Dynamic Mapped Statements (Section 3.9) can't use Parameter Maps. Being dynamic, the number of parameters will change and defeat the purpose of a Parameter Map. Depending on your provider, this may hinder your ability to use Dynamic Mapped Statements if your provider requires the use of some of the attributes, such as size or scale, that a <parameter> provides.

Parameter Maps can be provided as an external element and *inline*. Example 3.16 shows an external Parameter Map.

Example 3.16. An external Parameter Map

```
<parameterMap id="parameterMapIdentifier"
  [class="fullyQualifiedClassName, assembly|typeAlias"]
  [extends="[sqlMapNamespace.]parameterMapId"]>
  <parameter
    property ="propertyName"
    [column="columnName"]
    [direction="Input|Output|InputOutput"]
    [dbType="databaseType"]
    [type="propertyCLRType"]
    [nullValue="nullValueReplacement"]
    [size="columnSize"]
```

```
    [precision="columnPrecision"]
    [scale="columnScale"]
    [typeHandler="fullyQualifiedClassName, assembly|typeAlias"]
  <parameter ... ... />
  <parameter ... ... />
</parameterMap>
```

In Example 3.16, the parts in [brackets] are optional. The parameterMap element only requires the *id* attribute. The *class* attribute is optional but recommended. The *class* attribute helps to validate the incoming parameter and optimizes performance. Example 3.17 shows a typical <parameterMap>.

Example 3.17. A typical <parameterMap> element

```
<parameterMap id="insert-product-param" class="Product">
  <parameter property="description" />
  <parameter property="id"/>
</parameterMap>

<statement id="insertProduct" parameterMap="insert-product-param">
  insert into PRODUCT (PRD_DESCRIPTION, PRD_ID) values (?,?);
</statement>
```

Note

Parameter Map names are always local to the Data Map definition file where they are defined. You can refer to a Parameter Map in another Data Map definition file by prefixing the *id* of the Parameter Map with the namespace of the Data Map (set in the <sqlMap> root element). If the Parameter Map in Example 3.17 were in a Data Map named "Product", it could be referenced from another file using "Product.insert-product-param".

3.4.1. <parameterMap> attributes

The <parameterMap> element accepts three attributes: *id* (required), *class* (optional), and *extends* (optional).

3.4.1.1. id

The required id attribute provides a unique identifier for the <parameterMap> within this Data Map.

3.4.1.2. class

The optional *class* attribute specifies an object class to use with this <parameterMap>. The full classname and assembly or an alias must be specified. Any class can be used.

Note

The parameter class must be a property object or `IDictionary` instance.

3.4.1.3. extends

The optional *extends* attribute can be set to the name of another parameterMap upon which to base this parameterMap. All properties of the *super* parameterMap will be included as part of this

parameterMap, and values from the *super* parameterMap are set before any values specified by this parameterMap. The effect is similar to extending a class.

3.4.2. <parameter> Elements

The <parameterMap> element holds one or more parameter child elements that map object properties to placeholders in a SQL statement. The sections that follow describe each of the attributes.

3.4.2.1. property

The *property* attribute of <parameter> is the name of a field or property of the parameter object. It may also be the name of an entry in a `IDictionary` object. The name can be used more than once depending on the number of times it is needed in the statement. (In an update, you might set a column that is also part of the where clause.)

3.4.2.2. column

The *column* attribute is used to define to the name of a parameter used by a stored procedure.

3.4.2.3. direction

The *direction* attribute may be used to indicate a stored procedure parameter's direction.

Table 3.3. Parameter direction attribute values

Value	Description
Input	input-only
Output	output-only
InputOutput	bidirectional

3.4.2.4. dbType

The *dbType* attribute is used to explicitly specify the database column type of the parameter to be set by this property. For certain operations, some ADO.NET providers are not able to determine the type of a column, and the type must be specified.

This attribute is normally only required if the column is nullable. Although, another reason to use the dbType attribute is to explicitly specify date types. Whereas .NET only has one Date value type (`System.DateTime`), most SQL databases have more than one. Usually, a database has at least three different types (DATE, DATETIME, TIMESTAMP). In order for the value to map correctly, you might need to specify the column's dbType,

Note

Most providers only need the dbType specified for nullable columns. In this case, you only need to specify the type for the columns that are nullable.

The *dbType* attribute can be set to any string value that matches a constant in the specific data type enum of the used provider such as `System.Data.SqlDbType` for Microsoft Sql Server. Section 3.6

describes the types that are supported by the framework.

3.4.2.5. type

The *type* attribute is used to specify the CLR type of the parameter's *property*. This attribute is useful when passing InputOutput and Output parameters into stored procedures. The framework uses the specified *type* to properly handle and set the parameter object's properties with the procedure's output values after execution.

Normally, the type can be derived from a property through reflection, but certain mappings that use objects such as a Map cannot provide the property type to the framework. If the attribute type is not set and the framework cannot otherwise determine the type, the type is assumed to be an Object. Section 6 details the CLR types and available aliases that have pre-built support in the framework.

3.4.2.6. nullValue

The *nullValue* attribute can be set to any valid value (based on property type). The *nullValue* attribute is used to specify an outgoing null value replacement. What this means is that when the value is detected in the object property, a NULL will be written to the database (the opposite behavior of an inbound null value replacement). This allows you to use a *magic* null number in your application for types that do not support null values (such as `int`, `double`, `float`). When these types of properties contain a matching null value (–9999), a NULL will be written to the database instead of the value.

3.4.2.7. size

The *size* attribute sets the maximum size of the data within the column.

3.4.2.8. precision

The *precision* attribute is used to set the maximum number of digits used to represent the property value.

3.4.2.9. scale

The *scale* attribute sets the number of decimal places used to resolve the property value.

3.4.2.10. typeHandler

The *typeHandler* attribute allows the use of a Custom Type Handler (see the Custom Type Handler section). This allows you to extend the DataMapper's capabilities in handling types that are specific to your database provider, are not handled by your database provider, or just happen to be a part of your application design. You can create custom type handlers to deal with storing and retrieving booleans and Guids from your database for example.

3.4.3. Inline Parameter Maps

If you prefer to use inline parameters instead of parameter maps, you can add extra type information inline too. The inline parameter map syntax lets you embed the property name, the property type, the column type, and a null value replacement into a parametrized SQL statement. The next four examples shows statements written with inline parameters.

Example 3.18. A <statement> using inline parameters

```
<statement id="insertProduct" parameterClass="Product">
  insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
  values (#id#, #description#)
</statement>
```

The following example shows how dbTypes can be declared inline.

Example 3.19. A <statement> using an inline parameter map with a type

```
<statement id="insertProduct" parameterClass="Product">
  insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
  values (#id:int#, #description:VarChar#)
</statement>
```

The next example shows how dbTypes and null value replacements can also be declared inline.

Example 3.20. A <statement> using an inline parameter map with a null value replacement

```
<statement id="insertProduct" parameterClass="Product">
  insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
  values (#id:int:-999999#, #description:VarChar#)
</statement>
```

Like the DataMapper for Java, there is an alternate inline syntax that allows the specification of the property, type, dbType, and null value replacement. The following example shows that syntax in use.

Example 3.21. A <statement> using alternate inline syntax with property, type, dbType, and null value replacement

```
<update id="UpdateAccountViaInlineParameters" parameterClass="Account">
 update Accounts set
 Account_FirstName = #FirstName#,
 Account_LastName = #LastName#,
 Account_Email = #EmailAddress,type=string,dbType=Varchar,nullValue=no_email@provided.com#
 where
 Account_ID = #Id#
</update>
```

Note

When using inline parameters, you cannot specify the null value replacement without also specifying the dbType. You must specify both due to the parsing order.

For *round-trip* transparency of null values, you must also specify database columns null value replacements in your Result Map (see Section 3.5).

Inline parameter maps are handy for small jobs, but when there are a lot of type descriptors and null value replacements in a complex statement, an industrial-strength, external parameterMap can be easer.

3.4.4. Standard Type Parameters

In practice, you will find that many statements take a single parameter, often an `Integer` or a `String`. Rather than wrap a single value in another object, you can use the standard library object (`String`, `Integer`, et cetera) as the parameter directly. Example 3.22 shows a statement using a standard type parameter.

Example 3.22. A <statement> using standard type parameters

```
<statement id="getProduct" parameterClass="System.Int32">
  select * from PRODUCT where PRD_ID = #value#
</statement>
```

Assuming PRD_ID is a numeric type, when a call is made to this Mapped Statement, a standard Integer object can be passed in. The *#value#* parameter will be replaced with the value of the Integer instance. The name `value` is simply a placeholder, you can use another moniker of your choice. Result Maps support primitive types as results as well.

For your convenience, primitive types are aliased by the framework. For example, `int` can be used in place of `System.Integer`. For a complete list, see Section 3.6, "Supported Types for Parameter Maps and Result Maps".

3.4.5. Map or IDictionary Type Parameters

You can also pass a `IDictionary` instance as a parameter object. This would usually be a `HashTable`. Example 3.23 shows a <statement> using an `IDictionary` for a parameterClass.

Example 3.23. A <statement> using a Map or IDictionary for a parameterClass

```
<statement id="getProduct" parameterClass="System.Collections.IDictionary">
  select * from PRODUCT
  where PRD_CAT_ID = #catId#
  and PRD_CODE = #code#
</statement>
```

In Example 3.23, notice that the SQL in this Mapped Statement looks like any other. There is no difference in how the inline parameters are used. If a `HashTable` instance is passed, it must contain keys named catId and code. The values referenced by those keys must be of the appropriate type for the column, just as they would be if passed from a properties object.

For your convenience, `IDictionary` types are aliased by the framework. So, `map` or `HashTable` can be used in place of `System.Collections.Hashtable`. For a complete list of aliases, see Section 3.6, "Supported Types for Parameter Maps and Result Maps".

3.5. Result Maps

Section 3.4 describes Parameter Maps and Inline parameters, which map object properties to parameters in a database query. Result Maps finish the job by mapping the result of a database query (a set of columns) to object properties. Next to Mapped Statements, the Result Map is probably one of the most commonly used and most important features to understand.

A Result Map lets you control how data is extracted from the result of a query, and how the columns are mapped to object properties. A Result Map can describe the column type, a null value replacement, and complex property mappings including Collections. Example 3.24 shows the structure of a <resultMap> element.

Example 3.24. The structure of a <resultMap> element.

```
<resultMap id="resultMapIdentifier"
           [class="fullyQualifiedClassName, assembly|typeAlias"]
           [extends="[sqlMapNamespace.]resultMapId"]>

   <constructor >
       <argument property="argumentName"
           column="columnName"
           [columnIndex="columnIndex"]
           [dbType="databaseType"]
           [type="propertyCLRType"]
           [resultMapping="resultMapName"]
           [nullValue="nullValueReplacement"]
           [select="someOtherStatementName"]
           [typeHandler="fullyQualifiedClassName, assembly|typeAlias"] />
   </constructor >

   <result property="propertyName"
           column="columnName"
           [columnIndex="columnIndex"]
           [dbType="databaseType"]
           [type="propertyCLRType"]
           [resultMapping="resultMapName"]
           [nullValue="nullValueReplacement"]
           [select="someOtherStatementName"]
           [lazyLoad="true|false"]
           [typeHandler="fullyQualifiedClassName, assembly|typeAlias"]
   />
   <result ... .../>
   <result ... .../>
    // Inheritance support
   <discriminator column="columnName"
                    [type|typeHandler="fullyQualifiedClassName, assembly|typeAlias"]
   />
    <subMap value="discriminatorValue"
              resultMapping="resultMapName"
   />
   <subMap .../>
</resultMap>
```

In Example 3.24, the [brackets] indicate optional attributes. The *id* attribute is required and provides a name for the statement to reference. The class attribute is also required, and specifies a Type Alias or the fully qualified name of a class. This is the class that will be instantiated and populated based on the result mappings it contains.

The resultMap can contain any number of property mappings that map object properties to the columns of a result element. The property mappings are applied, and the columns are read, in the order that they are defined. Mainting the element order ensures consistent results between different drivers and providers.

Note

As with parameter classes, the result class must be a .NET object or IDictionary instance.

3.5.1. Extending resultMaps

The optional extends attribute can be set to the name of another resultMap upon which to base this resultMap. All properties of the "super" resultMap will be included as part of this resultMap, and values from the "super" resultMap are set before any values specified by this resultMap. The effect is similar to extending a class.

Tip

The "super" resultMap must be defined in the file *before* the extending resultMap. The classes for the super and sub resultMaps need not be the same, and do not need to be related in any way.

3.5.2. <resultMap> attributes

The <resultMap> element accepts three attributes: `id` (required), `class` (optional), and `extends` (optional).

3.5.2.1. id

The required `id` attribute provides a unique identifier for the <resultMap> within this Data Map.

3.5.2.2. class

The optional `class` attribute specifies an object class to use with this <resultMap>. The full classname or an alias must be specified. Any class can be used.

Note

As with parameter classes, the result classes must be a .NET object or IDictionary instance.

3.5.2.3. extends

The optional `extends` attribute allows the result map to inherit all of the properties of the "super" resultMap that it extends.

3.5.3. <constructor> element

The <constructor> element must match the signature of one of the result class constructor. If specify, this element is used by iBATIS to instanciate the result object.

The <constructor> element holds one or more <argument> child elements that map SQL resultsets to object argument constructor.

Example 3.25. Constructor element example

```
<resultMap id="account-result-constructor" class="Account" >
        <constructor>
               <argument argumentName="id" column="Account_ID"/>
```

```
                <argument argumentName="firstName" column="Account_FirstName"/>
                <argument argumentName="lastName" column="Account_LastName"/>
        </constructor>
        <result property="EmailAddress" column="Account_Email" nullValue="no_email@provided.com"/>
        <result property="BannerOption" column="Account_Banner_Option" dbType="Varchar" type="bool"/>
        <result property="CartOption" column="Account_Cart_Option" typeHandler="HundredsBool"/>
</resultMap>
```

3.5.3.1. argumentName

The *argumentName* attribute is the name of a constructor argument of the result object that will be returned by the Mapped Statement.

3.5.3.2. column

The *column* attribute value is the name of the column in the result set from which the value will be used to populate the argument.

3.5.3.3. columnIndex

As an optional (minimal) performance enhancement, the *columnIndex* attribute value is the index of the column in the ResultSet from which the value will be used to populate the object argument. This is not likely needed in 99% of applications and sacrifices maintainability and readability for speed. Some providers may not realize any performance benefit, while others will speed up dramatically.

3.5.3.4. dbType

The dbType attribute is used to explicitly specify the database column type of the ResultSet column that will be used to populate the argument. Although Result Maps do not have the same difficulties with null values, specifying the type can be useful for certain mapping types such as Date properties. Because an application language has one Date value type and SQL databases may have many (usually at least 3), specifying the date may become necessary in some cases to ensure that dates (or other types) are set correctly. Similarly, String types may be populated by a VarChar, Char or CLOB, so specifying the type might be needed in those cases too.

3.5.3.5. type

The type attribute is used to explicitly specify the CLR argument type. Normally this can be derived from a argument through reflection, but certain mappings that use objects such as a Map cannot provide the type to the framework. If the attribute type is not set and the framework cannot otherwise determine the type, the type is assumed to be Object. Section 6 details the CLR types and available aliases that are supported by the framework.

3.5.3.6. resultMapping

The *resultMapping* attribute can be set to the name of another resultMap used to fill the argument. If the resultMap is in an other mapping file, you must specified the fully qualified name.

3.5.3.7. nullValue

The *nullValue* attribute can be set to any valid value (based on argument type). The result element's *nullValue* attribute is used to specify an inbound null value replacement. What this means is that when the value is detected in a query's result column, the corresponding object argument will be set

to the the *nullValue* attribute's value. This allows you to use a "magic" null number in your application for types that do not support null values (such as `int`, `double`, `float`).

3.5.3.8. select

The select attribute is used to describe a relationship between objects and to automatically load complex (i.e. user defined) property types. The value of the statement property must be the name of another mapped statement. The value of the database column (the column attribute) that is defined in the same property element as this statement attribute will be passed to the related mapped statement as the parameter. More information about supported primitive types and complex property mappings/relationships is discussed later in this document. The lazyLoad attribute can be specified with the select

3.5.3.9. typeHandler

The typeHandler attribute allows the use of a Custom Type Handler (see the Custom Type Handler section). This allows you to extend the DataMapper's capabilities in handling types that are specific to your database provider, are not handled by your database provider, or just happen to be a part of your application design. You can create custom type handlers to deal with storing and retrieving booleans and Guids from your database for example.

3.5.4. <result> Elements

The <resultMap> element holds one or more <result> child elements that map SQL resultsets to object properties.

3.5.4.1. property

The *property* attribute is the name of a field or a property of the result object that will be returned by the Mapped Statement. The name can be used more than once depending on the number of times it is needed to populate the results.

3.5.4.2. column

The *column* attribute value is the name of the column in the result set from which the value will be used to populate the property.

3.5.4.3. columnIndex

As an optional (minimal) performance enhancement, the *columnIndex* attribute value is the index of the column in the ResultSet from which the value will be used to populate the object property. This is not likely needed in 99% of applications and sacrifices maintainability and readability for speed. Some providers may not realize any performance benefit, while others will speed up dramatically.

3.5.4.4. dbType

The dbType attribute is used to explicitly specify the database column type of the ResultSet column that will be used to populate the object property. Although Result Maps do not have the same difficulties with null values, specifying the type can be useful for certain mapping types such as Date properties. Because an application language has one Date value type and SQL databases may have many (usually at least 3), specifying the date may become necessary in some cases to ensure that dates (or other types) are set correctly. Similarly, String types may be populated by a VarChar, Char

or CLOB, so specifying the type might be needed in those cases too.

3.5.4.5. type

The type attribute is used to explicitly specify the CLR property type of the parameter to be set. Normally this can be derived from a property through reflection, but certain mappings that use objects such as a Map cannot provide the type to the framework. If the attribute type is not set and the framework cannot otherwise determine the type, the type is assumed to be Object. Section 6 details the CLR types and available aliases that are supported by the framework.

3.5.4.6. resultMapping

The *resultMapping* attribute can be set to the name of another resultMap used to fill the property. If the resultMap is in an other mapping file, you must specified the fully qualified name as :

```
resultMapping="[namespace.sqlMap.]resultMappingId"

resultMapping="Newspaper"
<!--resultMapping with a fully qualified name.-->
resultMapping="LineItem.LineItem"
```

3.5.4.7. nullValue

The *nullValue* attribute can be set to any valid value (based on property type). The result element's *nullValue* attribute is used to specify an inbound null value replacement. What this means is that when the value is detected in a query's result column, the corresponding object property will be set to the the *nullValue* attribute's value. This allows you to use a "magic" null number in your application for types that do not support null values (such as `int`, `double`, `float`).

If your database has a NULLABLE column, but you want your application to represent NULL with a constant value, you can specify it in the Result Map as shown in Example 3.25.

Example 3.26. Specifying a nullvalue attribute in a Result Map

```
<resultMap id="get-product-result" class="product">
  <result property="id" column="PRD_ID"/>
  <result property="description" column="PRD_DESCRIPTION"/>
  <result property="subCode" column="PRD_SUB_CODE" nullValue="-9999"/>
</resultMap>
```

In Example 3.25, if PRD_SUB_CODE is read as NULL, then the subCode property will be set to the value of -9999. This allows you to use a primitive type in your .NET class to represent a NULLABLE column in the database. Remember that if you want this to work for queries as well as updates/inserts, you must also specify the nullValue in the Parameter Map (discussed earlier in this document).

3.5.4.8. select

The select attribute is used to describe a relationship between objects and to automatically load complex (i.e. user defined) property types. The value of the statement property must be the name of another mapped statement. The value of the database column (the column attribute) that is defined in the same property element as this statement attribute will be passed to the related mapped statement as the parameter. More information about supported primitive types and complex property

mappings/relationships is discussed later in this document. The lazyLoad attribute can be specified with the select

3.5.4.9. lazyLoad

Use the lazyLoad attribute with the select attribute to indicate whether or not the select statement's results should be lazy loaded. This can provide a performance boost by delaying the loading of the select statement's results until they are needed/accessed. Lazy loading is supported transparently for IList and IList<T> implementation. Lazy loading is supported on strongly typed collection via Castle.DynamicProxy component. In this case you must set the listClass attribute and declare all methods/properties of the typed collection that you want to proxy as virtual.

Example 3.27. Sample of strongly typed collection used with proxy call

```
[C#]

[Serializable]
public class LineItemCollection : CollectionBase
{
        public LineItemCollection() {}

        public virtual LineItem this[int index]
        {
                get { return (LineItem)List[index]; }
                set { List[index] = value; }
        }

        public virtual int Add(LineItem value)
        {
                return List.Add(value);
        }

        public virtual void AddRange(LineItem[] value)
        {
                for (int i = 0; i < value.Length; i++)
                {
                        Add(value[i]);
                }
        }

        public virtual void AddRange(LineItemCollection value)
        {
                for (int i = 0; i < value.Count; i++)
                {
                        Add(value[i]);
                }
        }

        public virtual bool Contains(LineItem value)
        {
                return List.Contains(value);
        }

        public virtual void CopyTo(LineItem[] array, int index)
        {
                List.CopyTo(array, index);
        }

        public virtual int IndexOf(LineItem value)
        {
                return List.IndexOf(value);
        }

        public virtual void Insert(int index, LineItem value)
        {
                List.Insert(index, value);
        }
```

```
        public virtual void Remove(LineItem value)
        {
                List.Remove(value);
        }

        public new virtual int Count
        {
                get {return this.List.Count;}
        }
}
```

3.5.4.10. typeHandler

The typeHandler attribute allows the use of a Custom Type Handler (see the Custom Type Handler section). This allows you to extend the DataMapper's capabilities in handling types that are specific to your database provider, are not handled by your database provider, or just happen to be a part of your application design. You can create custom type handlers to deal with storing and retrieving booleans and Guids from your database for example.

3.5.5. Custom Type Handlers

A custom type handler allows you to extend the DataMapper's capabilities in handling types that are specific to your database provider, not handled by your database provider, or just happen to be part of your application design. The .NET DataMapper provides an interface, IBatisNet.DataMappers.TypeHandlers.ITypeHandlerCallback, for you to use in implementing your custom type handler.

Example 3.28. ITypeHandlerCallback interface

```
using System.Data;
using IBatisNet.DataMapper.Configuration.ParameterMapping;


namespace IBatisNet.DataMapper.TypeHandlers
{
 public interface ITypeHandlerCallback
 {
  void SetParameter(IParameterSetter setter, object parameter);

  object GetResult(IResultGetter getter);

  object ValueOf(string s);
 }
}
```

The `SetParameter` method allows you to process a <statement> parameter's value before it is added as an IDbCommand parameter. This enables you to do any necessary type conversion and clean-up before the DataMapper gets to work. If needed, you also have access to the underlying IDataParameter through the setter.DataParameter property.

The `GetResult` method allows you to process a database result value right after it has been retrieved by the DataMapper and before it is used in your resultClass, resultMap, or listClass. If needed, you also have access to the underlying IDataReader through the getter.DataReader property.

The `ValueOf` method allows you to compare a string representation of a value with one that you are

expecting and can handle appropriately. Typically, this is useful for translating a null value, but if your application or database will not support a null value, you can basically return the given string. When presented with an unexpected value, you can throw an appropriate exception.

One scenario that is familiar to .NET developers is the handling of a Guid type/structure. Many providers do not handle Guid class properties well, and developers may be faced with littering their domain objects with an additional set of property accessors to translate Guid properties into strings and strings into Guids.

```
public class BudgetObjectCode
{
 private string _code;
 private string _description;
 private Guid _guidProperty;

...

 public Guid GuidProperty {
  get { return _guidProperty; }
  set { _guidProperty = value; }
 }

 public string GuidPropertyString {
  get { return _guidProperty.ToString(); }
  set {
  if (value == null) {
    _guidProperty = Guid.Empty;
   }
   else {
    _guidProperty = new Guid(value.ToString());
   }
  }
 }
...
}
```

We can use a custom type handler to clean up this domain class. First, we define a string that will represent a null Guid value (Guid.Empty). We can then use that constant in our ValueOf null value comparison for the DataMapper to eventually use in setting our domain class' Guid properties. Implementing the GetResult and SetParameter methods is straightforward since we had been basically doing the same translation in our domain class' GuidPropertyString accessors.

Example 3.29. Guid String Type Handler

```
using System;
using IBatisNet.DataMapper.TypeHandlers;

namespace BigApp.Common.TypeHandlers
{
 /// <summary>
 /// GuidVarcharTypeHandlerCallback.
 /// </summary>
 public class GuidVarcharTypeHandlerCallback : ITypeHandlerCallback
 {
  private const string GUIDNULL = "00000000-0000-0000-0000-000000000000";

  public object ValueOf(string nullValue)
  {
   if (GUIDNULL.Equals(nullValue))
   {
    return Guid.Empty;
   }
   else
   {
    throw new Exception(
     "Unexpected value " + nullValue +
     " found where "+GUIDNULL+" was expected to represent a null value.");
   }
```

```
  }

  public object GetResult(IResultGetter getter)
  {
   try {
    Guid result = new Guid(getter.Value.ToString());
    return result;
   }
   catch
   {
     throw new Exception(
     "Unexpected value " + getter.Value.ToString() +
     " found where a valid GUID string value was expected.");
   }
  }

  public void SetParameter(IParameterSetter setter, object parameter)
  {
   setter.Value = parameter.ToString();
  }

 }
}
```

With our custom type handler, we can clean up our domain class and use the handler in our SqlMaps. To do that, we have two options in configuring our custom type handler to be used by the DataMapper. We can simply add it as a <typeAlias> and use it when needed in a parameterMap or resultMap.

Example 3.30. Aliased Custom Type Handler in a SqlMap.xml file

```
<alias>
 <typeAlias alias="GuidVarchar"
         type="BigApp.Common.TypeHandlers.GuidVarcharTypeHandlerCallback,
               BigApp.Common"/>
</alias>

<resultMaps>
 <resultMap id="boc-result"  class="BudgetObjectCode">
  <result property="Code" column="BOC_CODE" dbType="Varchar2"/>
  <result property="Description" column="BOC_DESC" dbType="Varchar2"/>
  <result property="GuidProperty" column="BOC_GUID" typeHandler="GuidVarchar"/>
 </resultMap>
</resultMaps>
```

Or we can specify it as a basic <typeHandler> for all Guid types mapped in our SqlMap files. <typeHandler> in SqlMap.config

Example 3.31. <typeHandler> in SqlMap.config

```
[Our SqlMap.config]
<alias>
 <typeAlias alias="GuidVarchar"
         type="BigApp.Common.TypeHandlers.GuidVarcharTypeHandlerCallback,
               BigApp.Common"/>
</alias>

<typeHandlers>
 <typeHandler type="guid" dbType="Varchar2" callback="GuidVarchar"/>
</typeHandlers>
```

```
[One of our SqlMap.xml files]
<parameterMaps>
 <parameterMap id="boc-params">
  <parameter property="Code" dbType="Varchar2" size="10"/>
  <parameter property="Description" dbType="Varchar2" size="100"/>
  <parameter property="GuidProperty" dbType="Varchar2" type="guid"/>
 </parameterMap>
</parameterMaps>

<resultMaps>
 <resultMap id="boc-result"  class="BudgetObjectCode">
  <result property="Code" column="BOC_CODE" dbType="Varchar2"/>
  <result property="Description" column="BOC_DESC" dbType="Varchar2"/>
  <result property="GuidProperty" column="BOC_GUID" dbType="Varchar2" type="guid"/>
 </resultMap>
</resultMaps>
```

3.5.6. Inheritance Mapping

The iBATIS DataMapper supports the implementation of object-oriented inheritance (subclassing) in your object model. There are several developer options for mapping entity classes and subclasses to database results:

- resultMap for each class
- resultMap with submaps for a class hierarchy
- resultMap with extended resultMaps for each subclass

You can use the most efficient mapping strategies from a SQL and query performance perspective when using the inheritance mappings of the DataMapper. To implement an inheritance mapping, the resultMap must define one or more columns in your query's resultset that will serve to identify which resultMap should be used to map each result record to a specific subclass. In many cases, you will use one column value for the DataMapper to use in identifying the proper resultMap and subclass. This column is known as a discriminator.

For example, we have a table defined in a database that contains Document records. There are five table columns used to store Document IDs, Titles, Types, PageNumbers, and Cities. Perhaps this table belongs to a legacy database, and we need to create an application using this table with a domain model that defines a class hierarchy of different types of Documents. Or perhaps we are creating a new application and database and just want to persist the data found in a set of related classes into one table. In either case, the DataMapper's inheritance mapping feature can help.

```
// Database table Document
CREATE TABLE [Documents] (
    [Document_ID] [int] NOT NULL ,
    [Document_Title] [varchar] (32) NULL ,
    [Document_Type] [varchar] (32)  NULL ,
    [Document_PageNumber] [int] NULL  ,
    [Document_City] [varchar] (32)  NULL
)
```

To illustrate this, let's take a look at a few example classes shown below that have a relationship through inheritance and whose properties can be persisted into our Documents table. First, we have a base Document class that has Id and Title properties. Next, we have a Book class that inherits from Document and contains an additional property called PageNumber. Last, we have a Newspaper class that also inherits from Document and contains a City property.

Example 3.32. Documents, Books, and Newspapers!

```
// C# class
public class Document
{
  private int _id = -1;
  private string _title = string.Empty;

  public int Id
  {
    get { return _id; }
    set { _id = value; }
  }

  public string Title
  {
    get { return _title; }
    set { _title = value; }
  }
}

public class Book : Document
{
  private int _pageNumber = -1;

  public int PageNumber
  {
    get { return _pageNumber; }
    set { _pageNumber = value; }
  }
}

public class Newspaper : Document
{
  private string _city = string.Empty;

  public string City
  {
    get { return _city; }
    set { _city = value; }
  }
}
```

Now that we have our classes and database table, we can start working on our mappings. We can create one <select> statement that returns all columns in the table. To help the DataMapper discriminate between the different Document records, we're going to indicate that the Document_Type column holds values that will distinguish one record from another for mapping the results into our class hierarchy.

```
// Document mapping file
<select id="GetAllDocument" resultMap="document">
   select
     Document_Id, Document_Title, Document_Type,
     Document_PageNumber, Document_City
   from Documents
   order by Document_Type, Document_Id
</select>

<resultMap id="document" class="Document">
  <result property="Id" column="Document_ID"/>
  <result property="Title" column="Document_Title"/>
  <discriminator column="Document_Type" type="string"/>
  <subMap value="Book" resultMapping="book"/>
  <subMap value="Newspaper" resultMapping="newspaper"/>
</resultMap>

<resultMap id="book" class="Book" extends="document">
  <property="PageNumber" column="Document_PageNumber"/>
</resultMap>
```

```
<resultMap id="newspaper" class="Newspaper"  extends="document">
  <property="City" column="Document_City"/>
</resultMap>
```

The DataMapper compares the data found in the discriminator column to the different <submap> values using the column value's string equivalence. Based on this string value, iBATIS DataMapper will use the resultMap named "Book" or "Newspaper" as defined in the <submap> elements or it will use the "super" resultMap "Document" if neither of the submap values satisfy the comparison. With these resultMaps, we can implement an object-oriented inheritance mapping to our database table.

If you want to use custom logic, you can use the typeHandler attribute of the <discriminator> element to specify a custom type handler for the discriminator column.

Example 3.33. Complex disciminator usage with Custom Type Handler

```
<alias>
  <typeAlias alias="CustomInheritance"
  type="IBatisNet.DataMapper.Test.Domain.CustomInheritance, IBatisNet.DataMapper.Test"/>
</alias>

<resultMaps>
  <resultMap id="document-custom-formula" class="Document">
    <result property="Id" column="Document_ID"/>
    <result property="Title" column="Document_Title"/>
    <discriminator column="Document_Type" typeHandler="CustomInheritance"/>
    <subMap value="Book" resultMapping="book"/>
    <subMap value="Newspaper" resultMapping="newspaper"/>
  </resultMap>
</resultMaps>
```

The value of the typeHandler attribute specifies which of our classes implements the *ITypeHandlerCallback* interface. This interface furnishes a GetResult method for coding custom logic to read the column result value and return a value for the DataMapper to use in its comparison to the resultMap's defined <submap> values.

Example 3.34. Example ITypeHandlerCallback interface implementation

```
public class CustomInheritance : ITypeHandlerCallback
{
  #region ITypeHandlerCallback members

  public object ValueOf(string nullValue)
  {
    throw new NotImplementedException();
  }

  public object GetResult(IResultGetter getter)
  {
   string type = getter.Value.ToString();

   if (type=="Monograph" || type=="Book")
   {
     return "Book";
   }
   else if (type=="Tabloid" || type=="Broadsheet" || type=="Newspaper")
   {
     return "Newspaper";
   }
   else
   {
     return "Document";
   }
```

```
  }

  public void SetParameter(IParameterSetter setter, object parameter)
  {
   throw new NotImplementedException();
  }
  #endregion
}
```

3.5.7. Implicit Result Maps

If the columns returned by a SQL statement match the result object, you may not need an explicit Result Map. If you have control over the relational schema, you might be able to name the columns so they also work as property names. In Example 3.33, the column names and property names already match, so a result map is not needed.

Example 3.35. A Mapped Statement that doesn't need a Result Map

```
<statement id="selectProduct" resultClass="Product">
  select
    id,
    description
  from PRODUCT
  where id = #value#
</statement>
```

Another way to skip a result map is to use column aliasing to make the column names match the properties names, as shown in Example 3.34.

Example 3.36. A Mapped Statement using column alaising instead of a Result Map

```
<statement id="selectProduct" resultClass="Product">
  select
  PRD_ID as id,
  PRD_DESCRIPTION as description
  from PRODUCT
  where PRD_ID = #value#
</statement>
```

Of course, these techniques will not work if you need to specify a column type, a null value, or any other property attributes.

Case sensitivity can also be an issue with implicit result maps. Conceivably, you could have an object with a "FirstName" property and a "Firstname" property. When iBATIS tries to match property and column, the heurstic is case-insensitive and we cannot guarantee which property would match. (Of course, very few developers would have two property names that were so simiilar.)

A final issue is that there is some performance overhead when iBATIS has to map the column and property names automatically. The difference can be dramatic if using a third-party NET database provider with poor support for ResultSetMetaData.

3.5.8. Primitive Results (i.e. String, Integer, Boolean)

Many times, we don't need to return an object with multiple properties. We just need a String, Integer, Boolean, and so forth. If you don't need to populate an object, iBATIS can return one of the primitive types instead. If you just need the value, you can use a standard type as a result class, as shown in Example 3.35.

Example 3.37. Selecting a standard type

```
<select id="selectProductCount" resultClass="System.Int32">
  select count(1)
  from PRODUCT
</select>
```

If need be, you can refer to the standard type using a marker token, "value", as shown by Example 3.36.

Example 3.38. Loading a simple list of product descriptions

```
<resultMap id="select-product-result" resultClass="System.String">
  <result property="value" column="PRD_DESCRIPTION"/>
</resultMap>
```

3.5.9. Maps with ResultMaps

Instead of a rich object, sometimes all you might need is a simple key/value list of the data, where each property is an entry on the list. If so, Result Maps can populate a IDictionary instance as easily as property objects. The syntax for using a IDictionary is identical to the rich object syntax. As shown in Example 3.37, only the result object changes.

Example 3.39. Result Maps can use generic "entry-type" objects

```
<resultMap id="select-product-result" class="HashTable">
  <result property="id" column="PRD_ID"/>
  <result property="code" column="PRD_CODE"/>
  <result property="description" column="PRD_DESCRIPTION"/>
  <result property="suggestedPrice" column="PRD_SUGGESTED_PRICE"/>
</resultMap>
```

In Example 3.37, an instance of HashTable would be created for each row in the result set and populated with the Product data. The property name attributes, like *id*, *code*, and so forth, would be the key of the entry, and the value of the mapped columns would be the value of the entry.

As shown in Example 3.38, you can also use an implicit Result Map with a IDictionary type.

Example 3.40. Implicit Result Maps can use "entry-type" objects too

```
<statement id="selectProductCount" resultClass="HashTable">
  select * from PRODUCT
</statement>
```

What set of entries is returned by Example xx depends on what columns are in the result set. If the set of column changes (because columns are added or removed), the new set of entries would automatically be returned.

Note

Certain providers may return column names in upper case or lower case format. When accessing values with such a provider, you will have to pass the Hashtable or HashMap key name in the expected case.

3.5.10. Complex Properties

In a relational database, one table will often refer to another. Likewise, some of your business objects may include another object or list of objects. Types that nest other types are called "complex types". You may not want a statement to return a simple type, but a fully-formed compex type.

In the database, a related column is usually represented via a 1:1 relationship, or a 1:M relationship where the class that holds the complex property is from the "many side" of the relationship and the property itself is from the "one side" of the relationship. The column returned from the database will not be the property we want; it is a key to be used in another query.

From the framework's perspective, the problem is not so much loading a complex type, but loading each "complex property". To solve this problem, you can specify in the Result Map a statement to run to load a given property. In Example 3.39, the "category" property of the "select-product-result" element is a complex property.

Example 3.41. A Result Map with a Complex Property

```
<resultMaps>
  <resultMap id="select-product-result" class="product">
    <result property="id" column="PRD_ID"/>
    <result property="description" column="PRD_DESCRIPTION"/>
    <result property="category" column="PRD_CAT_ID" select="selectCategory"/>
  </resultMap>

  <resultMap id="select-category-result" class="category">
    <result property="id" column="CAT_ID"/>
    <result property="description" column="CAT_DESCRIPTION"/>
  </resultMap>
</resultMaps>

<statements>
  <select id="selectProduct" parameterClass="int" resultMap="select-product-result">
   select * from PRODUCT where PRD_ID = #value#
  </select>

  <select id="selectCategory" parameterClass="int" resultMap="select-category-result">
   select * from CATEGORY where CAT_ID = #value#
  </select>
</statements>
```

In Example 3.39, the framework will use the "selectCategory" statement to populate the "category" property. The value of each category is passed to the "selectCategory" statement, and the object returned is set to the category property. When the process completes, each Product instance will have the the appropriate category object instance set.

3.5.11. Avoiding N+1 Selects (1:1)

A problem with Example 3.39 may be that whenever you load a Product, two statements execute: one for the Product and one for the Category. For a single Product, this issue may seem trivial. But if you load 10 products, then 11 statements execute. For 100 Products, instead of one statement product statement executing, a total of 101 statements execute. The number of statements executing for Example 3.40 will always be N+1: 100+1=101.

Example 3.42. N+1 Selects (1:1)

```
<resultMaps>
  <resultMap id="select-product-result" class="product">
    <result property="id" column="PRD_ID"/>
    <result property="description" column="PRD_DESCRIPTION"/>
    <result property="category" column="PRD_CAT_ID" select="selectCategory"/>
  </resultMap>

  <resultMap id="select-category-result" class="category">
    <result property="id" column="CAT_ID"/>
    <result property="description" column="CAT_DESCRIPTION"/>
  </resultMap>
</resultMaps>

<statements>
  <!-- This statement executes 1 time -->
  <select id="selectProducts" parameterClass="int" resultMap="select-product-result">
   select * from PRODUCT
  </select>

  <!-- This statement executes N times (once for each product returned above) -->
  <select id="selectCategory" parameterClass="int" resultMap="select-category-result">
   select * from CATEGORY where CAT_ID = #value#
  </select>
</statements>
```

One way to mitigate the problem is to cache the "selectCategory" statement . We might have a hundred products, but there might only be five categories. Instead of running a SQL query or stored procedure, the framework will return the category object from it cache. A 101 statements would still run, but they would not be hitting the database. (See Section 3.8 for more about caches.)

Another solution is to use a standard SQL join to return the columns you need from the another table. A join can bring all the columns we need over from the database in a single query. When you have a nested object, you can reference nested properties using a dotted notation, like "category.description".

Example 3.41 solves the same problem as Example 3.40, but uses a join instead of nested properties.

Example 3.43. Resolving complex properties with a join

```
<resultMaps>
  <resultMap id="select-product-result" class="product">
    <result property="id" column="PRD_ID"/>
    <result property="description" column="PRD_DESCRIPTION"/>
    <result property="category" resultMapping="Category.CategoryResult" />
  </resultMap>
</resultMaps>

<statements>
  <statement id="selectProduct" parameterClass="int" resultMap="select-product-result">
```

```
    select *
    from PRODUCT, CATEGORY
    where PRD_CAT_ID=CAT_ID
    and PRD_ID = #value#
  </statement>
</statements>
```

Lazy Loading vs. Joins (1:1)

It's important to note that using a join is not always better. If you are in a situation where it is rare to access the related object (e.g. the category property of the Product class) then it might actually be faster to avoid the join and the unnecessary loading of all category properties. This is especially true for database designs that involve outer joins or nullable and/or non-indexed columns. In these situations it might be better to use the sub-select solution with lazy loading enabled. The general rule of thumb is: use the join if you're more likely going to access the associated properties than not. Otherwise, only use it if lazy loading is not an option.

If you're having trouble deciding which way to go, don't worry. No matter which way you go, you can always change it without impacting your application source code. Example 3.40 and 3.41 result in exactly the same object graph and are loaded using the exact same method call from the application. The only consideration is that if you were to enable caching, then the using the separate select (not the join) solution could result in a cached instance being returned. But more often than not, that won't cause a problem (your application shouldn't be dependent on instance level equality i.e. "==").

3.5.12. Complex Collection Properties

It is also possible to load properties that represent lists of complex objects. In the database the data would be represented by a M:M relationship, or a 1:M relationship where the class containing the list is on the "one side" of the relationship and the objects in the list are on the "many side". To load an IList of objects, there is no change to the statement (see example above). The only difference required to cause the iBATIS DataMapper framework to load the property as an IList is that the property on the business object must be of type System.Collections.IList. For example, if a Category has a IList of Product instances, the mapping would look like this (assuming Category has a property called "ProductList" of System.Collections.IList.):

Example 3.44. Mapping that creates a list of complex objects

```
<resultMaps>

  <resultMap id="select-category-result" class="Category">
    <result property="Id" column="CAT_ID"/>
    <result property="Description" column="CAT_DESCRIPTION"/>
    <result property="ProductList" column="CAT_ID" select="selectProductsByCatId"/>
  </resultMap>

  <resultMap id="select-product-result" class="Product">
    <result property="Id" column="PRD_ID"/>
    <result property="Description" column="PRD_DESCRIPTION"/>
  </resultMap>
<resultMaps>

<statements>

  <statement id="selectCategory" parameterClass="int" resultMap="select-category-result">
```

```
    select * from CATEGORY where CAT_ID = #value#
  </statement>

  <statement id="selectProductsByCatId" parameterClass="int" resultMap="select-product-result">
    select * from PRODUCT where PRD_CAT_ID = #value#
  </statement>
</statements>
```

3.5.13. Avoiding N+1 Select Lists (1:M and M:N)

This is similar to the 1:1 situation above, but is of even greater concern due to the potentially large amount of data involved. The problem with the solution above is that whenever you load a Category, two SQL statements are actually being run (one for the Category and one for the list of associated Products). This problem seems trivial when loading a single Category, but if you were to run a query that loaded ten (10) Categories, a separate query would be run for each Category to load its associated list of Products. This results in eleven (11) queries total: one for the list of Categories and one for each Category returned to load each related list of Products (N+1 or in this case 10+1=11). To make this situation worse, we're dealing with potentially large lists of data.

Example 3.45. N+1 Select Lists (1:M and M:N)

```
<resultMaps>

  <resultMap id="select-category-result" class="Category">
    <result property="Id" column="CAT_ID"/>
    <result property="Description" column="CAT_DESCRIPTION"/>
    <result property="ProductList" column="CAT_ID" select="selectProductsByCatId"/>
  </resultMap>

  <resultMap id="select-product-result" class="Product">
    <result property="Id" column="PRD_ID"/>
    <result property="Description" column="PRD_DESCRIPTION"/>
  </resultMap>
<resultMaps>

<statements>

  <!-- This statement executes 1 time -->
  <statement id="selectCategory" parameterClass="int" resultMap="select-category-result">
    select * from CATEGORY where CAT_ID = #value#
  </statement>

  <!-- This statement executes N times (once for each category returned above)
       and returns a list of Products (1:M) -->
  <statement id="selectProductsByCatId" parameterClass="int" resultMap="select-product-result">
    select * from PRODUCT where PRD_CAT_ID = #value#
  </statement>
</statements>
```

1:N & M:N Solution? Currently the feature that resolves this issue not implemented, but the development discussions are active, and we expect it to be included in a future release.

Lazy Loading vs. Joins (1:M and M:N)

As with the 1:1 situation described previously, it's important to note that using a join is not always better. This is even more true for collection properties than it was for individual value properties due to the greater amount of data. If you are in a situation where it is rare to access the related object (e.g. the

ProductList property of the Category class) then it might actually be faster to avoid the join and the unnecessary loading of the list of products. This is especially true for database designs that involve outer joins or nullable and/or non-indexed columns. In these situations it might be better to use the sub-select solution with the lazy loading. The general rule of thumb is: use the join if you're more likely going to access the associated properties than not. Otherwise, only use it if lazy loading is not an option.

As mentioned earlier, if you're having trouble deciding which way to go, don't worry. No matter which way you go, you can always change it without impacting your .NET code. The two examples above would result in exactly the same object graph and are loaded using the exact same method call. The only consideration is that if you were to enable caching, then the using the separate select (not the join) solution could result in a cached instance being returned. But more often than not, that won't cause a problem (your application should not be dependent on instance level equality i.e. "==").

3.5.14. Composite Keys or Multiple Complex Parameters Properties

You might have noticed that in the above examples there is only a single key being used as specified in the resultMap by the column attribute. This would suggest that only a single column can be associated to a related mapped statement. However, there is an alternate syntax that allows multiple columns to be passed to the related mapped statement. This comes in handy for situations where a composite key relationship exists, or even if you simply want to use a parameter of some name other than #value#. The alternate syntax for the column attribute is simply {param1=column1, param2=column2, …, paramN=columnN}. Consider the example below where the PAYMENT table is keyed by both Customer ID and Order ID:

Example 3.46. Mapping a composite key

```
<resultMaps>
  <resultMap id="select-order-result" class="order">
    <result property="id" column="ORD_ID"/>
    <result property="customerId" column="ORD_CST_ID"/>
    ...
    <result property="payments" column="itemId=ORD_ID, custId=ORD_CST_ID"
      select="selectOrderPayments"/>
  </resultMap>
<resultMaps>

<statements>

  <statement id="selectOrderPayments" resultMap="select-payment-result">
    select * from PAYMENT
    where PAY_ORD_ID = #itemId#
    and PAY_CST_ID = #custId#
  </statement>
</statements>
```

Optionally you can just specify the column names as long as they're in the same order as the parameters. For example:

```
ORD_ID, ORD_CST_ID
```

As usual, this is a slight performance gain with an impact on readability and maintainability.

Important! Currently the iBATIS DataMapper framework does not automatically resolve circular relationships. Be aware of this when implementing parent/child relationships (trees). An easy workaround is to simply define a second result map for one of the cases that does not load the parent object (or vice versa), or use a join as described in the "N+1 avoidance" solutions.

Note

Result Map names are always local to the Data Map definition file that they are defined in. You can refer to a Result Map in another Data Map definition file by prefixing the name of the Result Map with the namespace of the SqlMap set in the <sqlMap> root element.

3.6. Supported Types for Parameter Maps and Result Maps

Table 3.4 shows the basic Supported Types for Parameter Maps and Result Maps for .NET. You can extend the framework's support for additional types by using a Custom Type Handler.

Note

Nuallable type from .NET 2.0 are fully supported by iBATIS.NET.

Table 3.4. Supported Types for Parameter Maps and Result Maps (.NET)

CLR Type	Object/Map Property Mapping	Result Class/Parameter Class**	Type Alias**
System.ArrayList	Yes	Yes	list
System.Boolean	Yes	Yes	Boolean, bool
System.Byte	Yes	Yes	Byte, byte
System.Char	Yes	Yes	Char, char
System.DateTime	Yes	Yes	dateTime, date
System.Decimal	Yes	Yes	Decimal, decimal
System.Double	Yes	Yes	Double, double
System.Guid	Yes	Yes	guid
System.Hashtable	Yes	Yes	map, hashmap, hashtable
System.Int16	Yes	Yes	Int16, short, Short
System.Int32	Yes	Yes	Int32, int, Int, integer, Integer
System.Int64	Yes	Yes	Int64, long, Long
System.SByte	Yes	Yes	SByte, sbyte
System.Single	Yes	Yes	Float, float, Single, single
System.String	Yes	Yes	String, string
System.TimeSpan	Yes	Yes	N/A
System.UInt16	Yes	Yes	Short, short

CLR Type	Object/Map Property Mapping	Result Class/Parameter Class**	Type Alias**
System.UInt32	Yes	Yes	Uint, uint
System.UInt64	Yes	Yes	Ulong, ulong
Nullable<bool>	Yes	Yes	bool?
Nullable<byte>	Yes	Yes	byte?
Nullable<char>	Yes	Yes	char?
Nullable<DateTime>	Yes	Yes	DateTime?
Nullable<decimal>	Yes	Yes	decimal?
Nullable<double>	Yes	Yes	double?
Nullable<Int16>	Yes	Yes	Int16?
Nullable<Int32>	Yes	Yes	Int32?
Nullable<Int64>	Yes	Yes	Int64?
Nullable<SByte>	Yes	Yes	SByte?
Nullable<Single>	Yes	Yes	Single?
Nullable<UInt16>	Yes	Yes	UInt16?
Nullable<UInt32>	Yes	Yes	UInt32?
Nullable<UInt64>	Yes	Yes	UInt64?

3.7. Supported database types for Parameter Maps and Result Maps

Table 3.5 shows the basic Supported DbTypes for Parameter Maps and Result Maps for .NET that come with the System.Data and System.Data.OracleClient assemblies. See the .NET Developer Guide section on configuring the DataMapper to work with your provider's DbTypes.

Note

Nuallable type from .NET 2.0 are fully supported by iBATIS.NET.

The .NET Framework data provider type of a Parameter object is inferred from the .NET Framework type of the Value of the Parameter object, or from the DbType of the Parameter object. The following table shows the inferred Parameter type based on the object passed as the Parameter value or the specified DbType. You may specify the type of a Parameter in a generic fashion by setting the DbType property of the Parameter object to a particular System.Data.DbType specific to your database.

Table 3.5. Supported DbTypes for Parameter Maps and Result Maps (.NET)

CLR Type	iBATIS support	SqlDbType	OleDbType	OdbcType	OracleType
Byte[]	Yes	Binary, Image, VarBinary	Binary, VarBinary	Binary, Image, VarBinary	Raw
Boolean, bool?	Yes	Bit	Boolean	Bit	Byte
Byte, byte?	Yes	TinyInt	-	TinyInt	Byte
DateTime, DateTime?	Yes	DateTime, SmallDateTime	Date	Date, DateTime, SmallDateTime, Time	DateTime
char, char?	Yes	Not supported	Char	Char	Byte
Decimal, decimal?	Yes	Decimal, Money, SmallMoney	Decimal, Currency, Numeric	Decimal, Numeric	Number
Double, double?	Yes	Float	Double	Double	Double
Guid, Guid?	Yes	UniqueIdentifier	Guid	UniqueIdentifier	Raw
Int16, Int16?	Yes	SmallInt	SmallIInt	SmallInt	Int16
Int32, Int32?	Yes	Int	Integer	Int	Int32
Int64, Int64?	Yes	BigInt	BigInt	BigInt	Number
Single, Single?	Yes	Real	Single	Real	Float
String	Yes	Char, Nchar, NVarchar, Text, VarChar	Char, VarChar	Char, NChar, NText, NVarChar, Text, VarChar	NVarChar, VarChar
TimeSpan	No	Not supported	DBTime	Time	DateTime
UInt16, UInt16?	yes	Int	-	-	UInt16
UInt32, UInt32?	yes	Decimal	-	-	UInt32
UInt64, UInt64?	yes	Decimal	-	-	Number

3.8. Cache Models

Some values in a database are know to change slower than others. To improve performance, many developers like to cache often-used data to avoid making unnecessary trips back to the database. iBATIS provides its own caching system, that you configure through a <cacheModel> element.

The results from a query Mapped Statement can be cached simply by specifying the cacheModel parameter in the statement tag (seen above). A cache model is a configured cache that is defined within your DataMapper configuration file. Cache models are configured using the cacheModel element as follows:

Example 3.47. Configuation a cache using the Cache Model element

```
<cacheModel id="product-cache" implementation="LRU" readOnly="true" serialize="false">
  <flushInterval hours="24"/>
  <flushOnExecute  statement="insertProduct"/>
  <flushOnExecute  statement="updateProduct"/>
  <flushOnExecute  statement="deleteProduct"/>
  <property name="CacheSize" value="100"/>
</cacheModel>
```

The cache model above will create an instance of a cache named "product-cache" that uses a Least Recently Used (LRU) implementation. The value of the type attribute is either a fully qualified class name, or an alias for one of the included implementations (see below). Based on the flush elements specified within the cache model, this cache will be flushed every 24 hours. There can be only one flush interval element and it can be set using hours, minutes, seconds or milliseconds. In addition the cache will be flushed whenever the insertProduct, updateProduct, or deleteProduct mapped statements are executed. There can be any number of "flush on execute" elements specified for a cache. Some cache implementations may need additional properties, such as the ‘cache-size' property demonstrated above. In the case of the LRU cache, the size determines the number of entries to store in the cache. Once a cache model is configured, you can specify the cache model to be used by a mapped statement, for example:

Example 3.48. Specifying a Cache Model from a Mapped Statement

```
<statement id="getProductList" cacheModel="product-cache">
  select * from PRODUCT where PRD_CAT_ID = #value#
</statement>
```

3.8.1. Read-Only vs. Read/Write

The framework supports both read-only and read/write caches. Read-only caches are shared among all users and therefore offer greater performance benefit. However, objects read from a read-only cache should not be modified. Instead, a new object should be read from the database (or a read/write cache) for updating. On the other hand, if there is an intention to use objects for retrieval and modification, a read/write cache is recommended (i.e. required). To use a read-only cache, set readOnly="true" on the cache model element. To use a read/write cache, set readOnly="false". The default is read-only (true).

3.8.2. Serializable Read/Write Caches

As you may agree, caching per-session as described above may offer little benefit to global application performance. Another type of read/write cache that can offer a performance benefit to the entire application (i.e. not just per session) is a serializable read/write cache. This cache will return different instances (copies) of the cached object to each session. Therefore each session can safely modify the instance returned. Realize the difference in semantics here, usually you would expect the same instance to be returned from a cache, but in this case you'll get a different one. Also note that every object stored by a serializable cache must be serializable. This means that you will have difficulty using both lazy loading features combined with a serializable cache, because lazy proxies are not serializable. The best way to figure out what combination of caching, lazy loading and table joining is simply to try it out. To use a serializable cache, set readOnly="false" and serialize="true". By default cache models are read-only and non-serializable. Read-only caches will not be serialized (there's no benefit).

3.8.3. Cache Implementation

The cache model uses a pluggable framework for supporting different types of caches. The choice of cache is specified in the "implementation" attribute of the cacheModel element as discussed above. The class name specified must be an implementation of the ICacheController interface, or one of the three aliases discussed below. Further configuration parameters can be passed to the implementation via the property elements contained within the body of the cacheModel. Currently there are 3 implementations included with the .NET distribution. These are as follows:

3.8.4. "MEMORY"

The MEMORY cache implementation uses reference types to manage the cache behavior. That is, the garbage collector effectively determines what stays in the cache or otherwise. The MEMORY cache is a good choice for applications that don't have an identifiable pattern of object reuse, or applications where memory is scarce.

The MEMORY implementation is configured as follows:

Example 3.49. Configuring a memory-type cache

```
<cacheModel id="product-cache" implementation="MEMORY" >
  <flushInterval hours="24"/>
  <flushOnExecute  statement="insertProduct"/>
  <flushOnExecute  statement="updateProduct"/>
  <flushOnExecute  statement="deleteProduct"/>
  <property name="Type" value="WEAK"/>
</cacheModel>
```

Only a single property is recognized by the MEMORY cache implementation. This property, named 'reference-type' must be set to a value of STRONG, SOFT, or WEAK.

The following table describes the different reference types that can be used for a MEMORY cache.

Table 3.6. Reference types that can be used for a MEMORY cache

Type	Description
WEAK (default)	This reference type is probably the best choice in most cases and is the default if the reference-type is not specified. It will increase performance for popular results, but it will absolutely release the memory to be used in allocating other objects, assuming that the results are not currently in use.
SOFT (currently Java only)	This reference type will reduce the likelihood of running out of memory in case the results are not currently in use and the memory is needed for other objects. However, this is not the most aggressive reference type in that regard and memory still might be allocated and made unavailable for more important objects.
STRONG	This reference type will guarantee that the results stay

Type	Description
	in memory until the cache is explicitly flushed (e.g. by time interval or flush on execute). This is ideal for results that are: 1) very small, 2) absolutely static, and 3) used very often. The advantage is that performance will be very good for this particular query. The disadvantage is that if the memory used by these results is needed, then it will not be released to make room for other objects (possibly more important objects).

3.8.5. "LRU"

The LRU cache implementation uses an Least Recently Used algorithm to determines how objects are automatically removed from the cache. When the cache becomes over full, the object that was accessed least recently will be removed from the cache. This way, if there is a particular object that is often referred to, it will stay in the cache with the least chance of being removed. The LRU cache makes a good choice for applications that have patterns of usage where certain objects may be popular to one or more users over a longer period of time (e.g. navigating back and forth between paginated lists, popular search keys etc.).

The LRU implementation is configured as follows:

Example 3.50. Configuring a LRU type cache

```
<cacheModel id="product-cache"  implementation="LRU" >
  <flushInterval hours="24"/>
  <flushOnExecute  statement="insertProduct"/>
  <flushOnExecute  statement="updateProduct"/>
  <flushOnExecute  statement="deleteProduct"/>
   <property name="CacheSize" value="100"/>
</cacheModel>
```

Only a single property is recognized by the LRU cache implementation. This property, named CacheSize must be set to an integer value representing the maximum number of objects to hold in the cache at once. An important thing to remember here is that an object can be anything from a single String instance to an ArrayList of object. So take care not to store too much in your cache and risk running out of memory!

3.8.6. "FIFO"

The FIFO cache implementation uses an First In First Out algorithm to determines how objects are automatically removed from the cache. When the cache becomes over full, the oldest object will be removed from the cache. The FIFO cache is good for usage patterns where a particular query will be referenced a few times in quick succession, but then possibly not for some time later.

The FIFO implementation is configured as follows:

Example 3.51. Configuring a FIFO type cache

```
<cacheModel id="product-cache" implementation="FIFO" >
  <flushInterval hours="24"/>
  <flushOnExecute  statement="insertProduct"/>
  <flushOnExecute  statement="updateProduct"/>
  <flushOnExecute  statement="deleteProduct"/>
  <property name="CacheSize" value="100"/>
</cacheModel>
```

Only a single property is recognized by the FIFO cache implementation. This property, named CacheSize must be set to an integer value representing the maximum number of objects to hold in the cache at once. An important thing to remember here is that an object can be anything from a single String instance to an ArrayList of object. So take care not to store too much in your cache and risk running out of memory

3.9. Dynamic SQL

A very common problem with working directly with ADO is dynamic SQL. It is normally very difficult to work with SQL statements that change not only the values of parameters, but which parameters and columns are included at all. The typical solution is usually a mess of conditional if-else statements and horrid string concatenations. The desired result is often a query by example, where a query can be built to find objects that are similar to the example object. The iBATIS DataMapper API provides a relatively elegant solution that can be applied to any mapped statement element. Here is a simple example:

Example 3.52. A simple dynamic select sttatement, with two possible outcomes

```
<select id="dynamicGetAccountList" cacheModel="account-cache" parameterClass="Account" resultMap="account-result
  select * from ACCOUNT
    <isGreaterThan prepend="and" property="Id" compareValue="0">
       where ACC_ID = #Id#
    </isGreaterThan>
  order by ACC_LAST_NAME
</select>
```

In the above example, there are two possible statements that could be created depending on the state of the Id property of the parameter object. If the Id parameter is greater than 0, then the statement will be created as follows:

```
select * from ACCOUNT where ACC_ID = ?
```

Or if the Id parameter is 0 or less, the statement will look as follows.

```
select * from ACCOUNT
```

The immediate usefulness of this might not become apparent until a more complex situation is encountered. For example, the following is a somewhat more complex example.

Example 3.53. A complex dynamic select statement, with 16 possible outcomes

```
<select id="dynamicGetAccountList" parameterClass="Account" resultMap="account-result" >
  select * from ACCOUNT
    <dynamic prepend="WHERE">
      <isNotNull prepend="AND" property="FirstName">
        ( ACC_FIRST_NAME = #FirstName#
        <isNotNull prepend="OR" property="LastName">
          ACC_LAST_NAME = #LastName#
        </isNotNull>
        )
      </isNotNull>
      <isNotNull prepend="AND" property="EmailAddress">
        ACC_EMAIL like #EmailAddress#
      </isNotNull>
      <isGreaterThan prepend="AND" property="Id" compareValue="0">
        ACC_ID = #Id#
      </isGreaterThan>
    </dynamic>
  order by ACC_LAST_NAME
</select>
```

Depending on the situation, there could be as many as 16 different SQL queries generated from the above dynamic statement. To code the if-else structures and string concatenations could get quite messy and require hundreds of lines of code.

Using dynamic statements is as simple as inserting some conditional tags around the dynamic parts of your SQL. For example:

Example 3.54. Creating a dynamic statement with conditional tags

```
<statement id="someName" parameterClass="Account" resultMap="account-result" >
  select * from ACCOUNT
  <dynamic prepend="where">
    <isGreaterThan prepend="and" property="id" compareValue="0">
      ACC_ID = #id#
    </isGreaterThan>
    <isNotNull prepend="and" property="lastName">
      ACC_LAST_NAME = #lastName#
    </isNotNull>
  </dynamic>
order by ACC_LAST_NAME
</statement>
```

In the above statement, the <dynamic> element demarcates a section of the SQL that is dynamic. The dynamic element is optional and provides a way to manage a prepend in cases where the prepend ("WHERE") should not be included unless the contained conditions append to the statement. The statement section can contain any number of conditional elements (see below) that will determine whether the contained SQL code will be included in the statement. All of the conditional elements work based on the state of the parameter object passed into the query. Both the dynamic element and the conditional elements have a "prepend" attribute. The prepend attribute is a part of the code that is free to be overridden by the a parent element's prepend if necessary. In the above example the "where" prepend will override the first true conditional prepend. This is necessary to ensure that the SQL statement is built properly. For example, in the case of the first true condition, there is no need for the AND, and in fact it would break the statement. The following sections describe the various kinds of elements, including Binary Conditionals, Unary Conditionals, and Iterate.

3.9.1. Binary Conditional Elements

Binary conditional elements compare a property value to a static value or another property value. If the result is true, the body content is included in the SQL query.

3.9.1.1. Binary Conditional Attributes:

prepend – the overridable SQL part that will be prepended to the statement (optional)
property – the property to be compared (required)
compareProperty – the other property to be compared (required or compareValue)
compareValue – the value to be compared (required or compareProperty)

Table 3.7. Binary conditional attributes

Element	Description

<isEqual>	Checks the equality of a property and a value, or another property. Example Usage:
	`<isEqual prepend="AND"`
	`property="status"`
	`compareValue="Y">`
	`MARRIED = 'TRUE'`
	`</isEqual>`

<isNotEqual>	Checks the inequality of a property and a value, or another property. Example Usage:
	`<isNotEqual prepend="AND"`
	`property="status"`
	`compareValue="N">`
	`MARRIED = 'FALSE'`
	`</isNotEqual>`

<isGreaterThan>	Checks if a property is greater than a value or another property. Example Usage:
	`<isGreaterThan prepend="AND"`
	`property="age"`
	`compareValue="18">`
	`ADOLESCENT = 'FALSE'`
	`</isGreaterThan>`

<isGreaterEqual>	Checks if a property is greater than or equal to a value or another property. Example Usage:
	`<isGreaterEqual prepend="AND"`
	`property="shoeSize"`
	`compareValue="12">`
	`BIGFOOT = 'TRUE'`
	`</isGreaterEqual>`

<isLessEqual>	Checks if a property is less than or equal to a value or another property. Example Usage:
	`<isLessEqual prepend="AND"`
	`property="age"`
	`compareValue="18">`
	`ADOLESCENT = 'TRUE'`
	`</isLessEqual>`

3.9.2. Unary Conditional Elements

Unary conditional elements check the state of a property for a specific condition.

3.9.2.1. Unary Conditional Attributes:

prepend – the overridable SQL part that will be prepended to the statement (optional)
property – the property to be checked (required)

Table 3.8. Unary conditional attributes

Element	Description
<isPropertyAvailable>	Checks if a property is available (i.e is a property of the parameter object). Example Usage: `<isPropertyAvailable property="id" >   ACCOUNT_ID=#id# </isPropertyAvailable>`
<isNotPropertyAvailable>	Checks if a property is unavailable (i.e not a property of the parameter object). Example Usage: `<isNotPropertyAvailable property="age" >   STATUS='New' </isNotEmpty>`
<isNull>	Checks if a property is null. Example Usage: `<isNull prepend="AND" property="order.id" >   ACCOUNT.ACCOUNT_ID = ORDER.ACCOUNT_ID(+) </isNotEmpty>`
<isNotNull>	Checks if a property is not null. Example Usage: `<isNotNull prepend="AND" property="order.id" >   ORDER.ORDER_ID = #order.id# </isNotEmpty>`
<isEmpty>	Checks to see if the value of a Collection, String property is null or empty ("" or size() < 1). Example Usage: `<isEmpty property="firstName" >   LIMIT 0, 20 </isNotEmpty>`
<isNotEmpty>	Checks to see if the value of a Collection, String property is not null and not empty ("" or size() < 1). Example Usage: `<isNotEmpty prepend="AND" property="firstName" >   FIRST_NAME LIKE '%$FirstName$%' </isNotEmpty>`

3.9.3. Parameter Present Elements

These elements check for parameter object existence.

3.9.3.1. Parameter Present Attributes:

prepend – the overridable SQL part that will be prepended to the statement (optional)

Table 3.9. Testing to see if a parameter is present

Element	Description

<isParameterPresent>	Checks to see if the parameter object is present (not null).
	`<isParameterPresent prepend="AND">`
	`  EMPLOYEE_TYPE = #empType#`
	`</isParameterPresent>`

<isNotParameterPresent>	Checks to see if the parameter object is not present (null). Example Usage:
	`<isNotParameterPresent prepend="AND">`
	`  EMPLOYEE_TYPE = 'DEFAULT'`
	`</isNotParameterPresent>`

3.9.4. Iterate Element

This tag will iterate over a collection and repeat the body content for each item in a List

3.9.4.1. Iterate Attributes:

prepend – the overridable SQL part that will be prepended to the statement (optional)
property – a property of type IList that is to be iterated over (required)
open – the string with which to open the entire block of iterations, useful for brackets (optional)
close – the string with which to close the entire block of iterations, useful for brackets (optional)
conjunction – the string to be applied in between each iteration, useful for AND and OR (optional)

Table 3.10. Creating a list of conditional clauses

Element	Description

<iterate>	Iterates over a property that is of type IList Example Usage:
	`<iterate prepend="AND" property="UserNameList"`
	`  open="(" close=")" conjunction="OR">`
	`  username=#UserNameList[]#`
	`</iterate>`
	Note: It is very important to include the square brackets[] at the end of the List property name when using the Iterate element. These brackets distinguish

Element	Description
	this object as an List to keep the parser from simply outputting the List as a string.

3.9.5. Simple Dynamic SQL Elements

Despite the power of the full Dynamic Mapped Statement API discussed above, sometimes you just need a simple, small piece of your SQL to be dynamic. For this, SQL statements and statements can contain simple dynamic SQL elements to help implement dynamic order by clauses, dynamic select columns or pretty much any part of the SQL statement. The concept works much like inline parameter maps, but uses a slightly different syntax. Consider the following example:

Example 3.55. A dynamic element that changes the collating order

```
<statement id="getProduct" resultMap="get-product-result">
  select * from PRODUCT order by $preferredOrder$
</statement>
```

In the above example the preferredOrder dynamic element will be replaced by the value of the preferredOrder property of the parameter object (just like a parameter map). The difference is that this is a fundamental change to the SQL statement itself, which is much more serious than simply setting a parameter value. A mistake made in a Dynamic SQL Element can introduce security, performance and stability risks. Take care to do a lot of redundant checks to ensure that the simple dynamic SQL elements are being used appropriately. Also, be mindful of your design, as there is potential for database specifics to encroach on your business object model. For example, you may not want a column name intended for an order by clause to end up as a property in your business object, or as a field value on your server page.

Simple dynamic elements can be included within <statements> and come in handy when there is a need to modify the SQL statement itself. For example:

Example 3.56. A dynamic element that changes the comparison operator

```
<statement id="getProduct" resultMap="get-product-result">
  SELECT * FROM PRODUCT
  <dynamic prepend="WHERE">
    <isNotEmpty property="Description">
       PRD_DESCRIPTION $operator$ #Description#
    </isNotEmpty>
  </dynamic>
</statement>
```

In the above example the operator property of the parameter object will be used to replace the $operator$ token. So if the operator property was equal to LIKE and the description property was equal to %dog%, then the SQL statement generated would be:

```
  SELECT * FROM PRODUCT WHERE PRD_DESCRIPTION LIKE '%dog%'
```

Chapter 4. .NET Developer Guide

4.1. Introduction

This section explains how to install, configure, and use the iBATIS DataMapper with your .NET application. It is assumed that you are using Microsoft Visual Studio .NET (VSN). If you are using another IDE, please modify these instructions accordingly.

4.2. Installing the DataMapper for .NET

There are four steps to using iBATIS DataMapper with your application for the first time.

1. Setup the distribution
2. Add assembly references
3. Visual Studio.NET Integration
4. Add XML documents

4.2.1. Setup the Distribution

The official site for iBATIS DataMapper for .NET is our Apache site <http://ibatis.apache.org/>. The DataMapper is availabe in 2 types of distributions: a binary distribution that includes the required DataMapper assemblies and a source distribution that includes a VSN solution. To download either of the distributions, follow the link to the Downloads area on our web site, and select the either the binary or source distribution for the iBATIS .NET DataMapper release (if you download the binary distribution, extract the files using a utility like WinZip or the extractor built into newer versions of Windows and skip ahead to the Add Assembly References section).

The DataMapper source distribution includes a VSN solution and a number of C# projects. The distribution is in the form of a ZIP archive. You can extract the distribution using a utility like WinZip or the extractor built into newer versions of Windows. We suggest that you create an `ibatisnet` folder in your VSN project directory and extract the distribution there.

Under the distribution's `source` folder are eight folders that make up the iBATIS.NET distribution, as shown in Table 4.1.

Table 4.1. Folders found in the iBATIS.NET source distribution

Folder name	Description
External-Bin	Dependency assemblies provided for your convenience.
IBatisNet.Common	Assembly of classes shared by DataAccess and DataMapper
IBatisNet.Common.Logging.Log4Net	Log4Net factory adapter classes
IBatisNet.Common.Test	Test project for IBatisNet.Common that can be used

Folder name	Description
	with NUnit
IBatisNet.DataAccess	The Data Access Objects framework (see separate DAO Guide)
IBatisNet.DataAccess.Extensions	Contains a C# project for extensions to the DataAccess framework such as NHibernate support
IBatisNet.DataAccess.Test	Test project for the DataAccess framework that can be used with NUnit
iBatisNet.DataMapper	The DataMapper framework
IBatisNet.DataMapper.Test	Test project for the DataMapper that can be used with NUnit

You can load the `IBatisNet.sln` solution file into VSN and build the solution to generate the needed assemblies. There are seven projects in the solution, and all should succeed. The assemblies we need will be created under `\source\IBatisNet.DataMapper\bin\Debug`. The created assemblies are :

1. `IBatisNet.Common.dll`
2. `iBatisNet.DataMapper.dll`

The DataMapper has external dependencies on :

1. `Castle.DynamicProxy.dll` (creating proxies)

This dependencies can be found in the External-Bin folder and/or in the bin\Debug folder after building the solution.

Tip

If you will not be using the DataAccess framework and NHibernate and you have a problem building the solution due to the dependency on NHibernate, simply remove the IBatisNet.DataAccess.Extensions and IBatisNet.DataAccess.Test projects from the solution before building.

4.2.2. Add Assembly References

Switching to your own solution, open the project that will be using the iBATIS .NET DataMapper. Depending on how you organize your solutions, this may or may not be the project for your Windows or Web application. It may be a library project that your application project references. You will need to add one or two references to your project:

1. IBatisNet.DataMapper.dll
2. IBatisNet.DataAccess.dll (optional)
3. IBatisNet.Common.dll (implied)
4. Castle.DynamicProxy.dll (implied)

If you are using the `Mapper` singleton (see section 4.4.1), then the only reference you will need is to the DataMapper assembly. The Common and Castle.DynamicProxy assemblies are needed at runtime, but Visual Studio.NET will resolve the dependencies for you. If you are using the Data Access Objects framework, then you will need a reference to the DataAccess assembly too. So, start with the first, and add the others only if needed.

If you have built the IBatisNet solution as described in Section 4.2.1, the three assemblies (IBatisNet.DataMapper.dll, IBatisNet.Common.dll, and Castle.DynamicProxy.dll) that you will need should be in the `bin/Debug` folder of the IBatisNet.DataMapper project.

4.2.3. Add XML File Items

After adding the assembly references, you will need to add three types of XML files to your Windows, Web application, or library project (and Test project if you have one). These files are:

- providers.config - A file used by the DataMapper to look up the definition of your selected database provider.
- SqlMap.xml - A Data Map file that contains your SQL queries. Your project will contain one or more of these files with names such as Account.xml or Product.xml.
- SqlMap.config - The DataMapper configuration file that is used to specify the locations of your SqlMap.xml files and providers.config file. It is also used to define other DataMapper configuration options such as caching. You will need to include one SqlMap.config file for each data source that your project has.

As expected, the `SqlMap.config` and `providers.config` files must be placed where the DataMapper can find them at runtime. Depending on the type of project you have, the default expected location of these 2 files will be different, as shown in Table 4.2. However, your project is not limited to using just these locations. The DataMapper provides other options for placing these files in locations that are more suitable for your project instead of using the default locations. These options are covered later in this guide.

Table 4.2. Default locations for the sqlMap.config and providers.config files

Windows, Library, or Test projects (using NUnit or equivalent)	This would be the binary folder (such as /bin/debug) with the assembly (.dll) files and the `App.config` file
Web projects	In the application root, where the `Web.config` file is located.

4.2.4. Visual Studio.NET Integration

Each configuration file (SqlMap.config, mappping file, providers.config) is associated to a schema. The benefits of associating an XML document with a schema are to validate the document (which is done at runtime) and to use editing features such as IntelliSense/content completion assistance.

To allow association of the schemas in VS.NET XML editor to yours configuration files, you should add the schema files (`SqlMap.xsd`, `SqlMapConfig.xsd`, `providers.xsd`) to either your VS.NET project or in your VS.NET installation directory. The VS.NET directory will be either

```
C:\Program Files\Microsoft Visual Studio 8\Xml\Schemas for VS.NET 2005
```

or

```
C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\Packages\schemas\xml for
VS.NET 2003
```

or

```
C:\Program Files\Microsoft Visual Studio .NET\Common7\Packages\schemas\xml for
VS.NET 2002
```

depending on your version of VS.NET. It is typically easier to place the file in the well known location under the VS.NET installation directory than to copy the XSD file for each project you create.

Once you have registered the schema with VS.NET you will be enough to get IntelliSense and validation of the configuration file from within VS.NET.

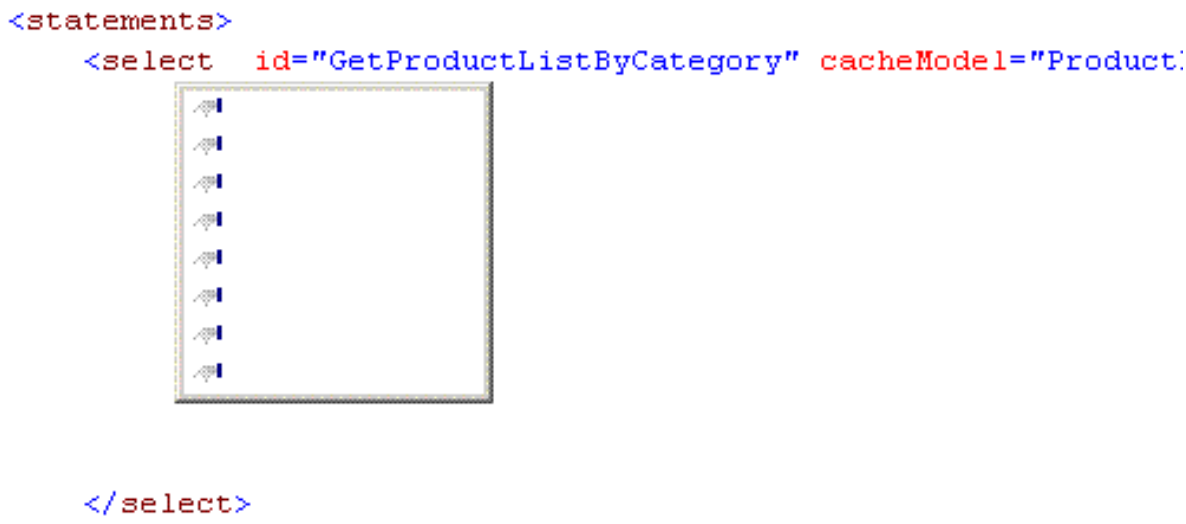

Figure 4.1. IntelliSense example

4.3. Configuring the DataMapper for .NET

The iBATIS DataMapper is configured using a central XML descriptor file, usually named `SqlMap.config`, which provides the details for your data source, data maps, and other features like caching, transactions, and thread management. At runtime, your application code will call a class method provided by the iBATIS library to read and parse your `SqlMap.config` file. After parsing the configuration file, a DataMapper client will be returned by iBATIS for your application to use.

4.3.1. DataMapper clients

Currently, the DataMapper framework revolves around the `SqlMapper` class, which acts as a facade to the DataMapper framework API. You can create a DataMapper client by instantiating an object of the `SqlMapper` class. An instance of the `SqlMapper` class (your DataMapper client) is created by reading a single configuration file. Each configuration file can specify one database or data source. However, you can use multiple DataMapper clients in your application. Just create another

configuration file and pass the name of that file when the DataMapper client is created. The configuration files might use a different account with the same database, or reference different databases on different servers. You can read from one client and write to another, if that's what you need to do. See Section 4.4.1 for more details on building a `SqlMapper` instance, but first, let's take a look at the DataMapper configuration file.

4.3.2. DataMapper Configuration File (SqlMap.config)

A sample configuration file for a .NET web application is shown in Example 4.1. Not all configuration elements are required. The following sections describe the elements of this SqlMap.config file in more detail.

Example 4.1. Sample SqlMap.Config for a .NET Web Application (placed in same directory as web.config)

```
<?xml version="1.0" encoding="utf-8"?>
<sqlMapConfig xmlns="http://ibatis.apache.org/dataMapper"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >

  <!-- Optional -->
  <properties resource="properties.config"/>

  <settings>
    <setting useStatementNamespaces="false"/>
    <setting cacheModelsEnabled="true"/>
    <setting validateSqlMap="false"/>
  </settings>

  <!-- Not required if providers.config is located in default location -->
  <providers embedded="resources.providers.config, IBatisNet.Test"/>

  <database>
    <provider name="sqlServer1.1"/>
    <dataSource name="NPetshop"
                connectionString="user id=${username};
                password=${password};
                data source=${datasource};
                database=${database};"/>
  </database>

  <alias>
    <typeAlias alias="Account" type="IBatisNet.Test.Domain.Account, IBatisNet.Test"/>
    <typeAlias alias="YesNoBool"
         type="IBatisNet.Test.Domain.YesNoBoolTypeHandlerCallback, IBatisNet.Test"/>
  </alias>

  <typeHandlers>
    <typeHandler type="bool" dbType="Varchar" callback="YesNoBool"/>
  </typeHandlers>

  <sqlMaps>
    <sqlMap resource="${root}Maps/Account.xml"/>
    <sqlMap resource="${root}Maps/Category.xml"/>
    <sqlMap resource="${root}Maps/Product.xml"/>
  </sqlMaps>
</sqlMapConfig>
```

4.3.3. DataMapper Configuration Elements

4.3.3.1. The <properties> Element

Sometimes the values we use in an XML configuration file occur in more than one element. Often,

there are values that change when we move the application from one server to another. To help you manage configuration values, you can specify a standard properties file (with name=value entries) as part of a DataMapper configuration. Each named value in the properties file becomes a *shell* variable that can be used in the DataMapper configuration file and your Data Map definition files (see Section 3). For example, if the "properties" file contains

```
<?xml version="1.0" encoding="utf-8" ?>
<settings>
  <add key="username" value="albert" />
</settings>
```

then all elements in the DataMapper configuration can use the variable `${username}` to insert the value "*albert*". For example:

```
<dataSource connectionString="user id=${username};"
```

Properties are handy during building, testing, and deployment by making it easy to reconfigure your application for multiple environments or use automated tools for configuration such as NAnt.

4.3.3.1.1. <properties> attributes

The <properties> element can accept one of the following attributes to specify the location of the properties file.

Table 4.3. Attributes of the <properties> element

Attribute	Description
resource	Specify the properties file to be loaded from the root directory of the application `resource="properties.config"`

url	Specify the properties file to be loaded through an absolute path.
	`url="c:\Web\MyApp\Resources\properties.config"`
	`-or-`
	`url="file://c:\Web\MyApp\Resources\properties.config"`

embedded	Specify the properties file to be loaded as an embedded resource in an assembly. Syntax for the embedded attribute is '*[extendednamespace.]filename, the name of the assembly which contains the embedded resource*' `embedded="Resources.properties.config, MyApp.Data"`

4.3.3.1.2. <property> element and attributes

You can also specify more than one properties file or add property keys and values directly into your

`SqlMap.config` file by using <property> elements. For example:

```
<properties>
 <property resource="myProperties.config"/>
 <property resource="anotherProperties.config"/>
 <property key="host" value="ibatis.com" />
</properties>
```

Table 4.4. Attributes of the <property> element

Attribute	Description
resource	Specify the properties file to be loaded from the root directory of the application `resource="properties.config"`

url	Specify the properties file to be loaded through an absolute path.
	`url="c:\Web\MyApp\Resources\properties.config"`
	`-or-`
	`url="file://c:\Web\MyApp\Resources\properties.config"`

embedded	Specify the properties file to be loaded as an embedded resource in an assembly. Syntax for the embedded attribute is '*[extendednamespace.]filename, the name of the assembly which contains the embedded resource*' `embedded="Resources.properties.config, MyApp.Data"`
key	Defines a property key (variable) name `key="username"`
value	Defines a value that will be used by the DataMapper in place of the the specified property key/variable `value="mydbuser"`

4.3.3.2. The <providers> Element

Under ADO.NET, a database system is accessed through a provider. A database system can use a custom provider or a generic ODBC provider. The iBATIS .NET DataMapper uses a pluggable approach to using providers. Each provider is represented by an XML descriptor element found in a file called `providers.config`. The iBATIS .NET DataMapper distribution includes a standard `providers.config` file with a set of thirteen prewritten provider elements:

- sqlServer1.0 - Microsoft SQL Server 7.0/2000 provider available with .NET Framework 1.0

- sqlServer1.1 -Microsoft SQL Server 7.0/2000 provider available with .NET Framework 1.1
- OleDb1.1 - OleDb provider available with .NET Framework 1.1
- Odbc1.1 - Odbc provider available with .NET Framework 1.1
- sqlServer2.0 -Microsoft SQL Server 7.0/2000/2005 provider available with .NET Framework 2.0
- OleDb2.0 - OleDb provider available with .NET Framework 2.0
- Odbc2.0 - Odbc provider available with .NET Framework 2.0
- oracle9.2 - Oracle provider V9.2.0.401
- oracle10.1 - Oracle provider V10.1.0.301
- oracleClient1.0 - MS Oracle provider V1.0.5 available with .NET Framework 1.1
- ByteFx - ByteFx MySQL provider V0.7.6.15073
- MySql - MySQL provider V1.0.4.20163
- SQLite3 - SQLite.NET provider V0.21.1869.3794
- Firebird1.7 - Firebird SQL .NET provider V1.7.0.33200
- PostgreSql0.7 - Npgsql provider V0.7.0.0
- iDb2.10 - IBM DB2 iSeries provider V10.0.0.0

Note

If you use SQL Server 2005, you can configure the provider to allow Multiple Active Result Set (allowMARS="true") and add MultipleActiveResultSets=true in your connection string.

The `providers.config` file can be found under `\source\IBatisNet.DataMapper.Test\bin\Debug` in the iBATIS .NET source distribution or in the root folder of the .NET DataMapper binary distribution.

A provider may require libraries that you do not have installed,. Therefore, the provider element has an "enabled" attribute that allows you to disable unused providers. One provider can also be marked as the "default" and will be used if another is not specified by your configuration.

The standard `providers.config` file has sqlServer1.1 set as the default and the sqlServer1.0 provider disabled. Aside from sqlServer1.1, OleDb1.1, and Odbc1.1, all other providers are disabled by default. Remember to set the "enabled" attribute to "true" for the provider that you will be using.

Important

ByteFx is the recommended provider if you are using MySQL. You may download ByteFx from the MySQLNet SourceForge site (http://sf.net/projects/mysqlnet/). If the ByteFx license is acceptable to you, you may install it as a reference within your application and enable the ByteFx provider.

Tip

Be sure to review the `providers.config` file and confirm that the provider you intend to use is enabled by setting the "enabled" attribute to "true".

Table 4.5. Expected default locations of the providers.config file

Windows, Library, or Test projects (using NUnit or equivalent)	With the assembly (.dll) files with the `app.config` file
Web projects	In the project base directory, with the `web.config` file

To use the file, you can copy it into your project at the expected default location, give a path to the file relative to the project root directory, specify a url (absolute path) to its location, or make it an embedded resource of your project. If you copy the file into the expected default location, the <providers> element is not required in your sqlMap.config file.

4.3.3.2.1. <providers> attributes

The <providers> element can accept one of the following attributes to specify the location of the providers.config file.

Table 4.6. Attributes of the <providers> element

Attribute	Description
resource	Specify the file to be loaded from a relative path from the project root directory. Since the root directory is different depending on the project type, it is best to use a properties variable to indicate the relative path. Having that variable defined in a properties file makes it easy to change the path to all your Data Mapper configuration resources in one location. `resource="${root}providers.config"`

url	Specify the providers.config to be loaded through an absolute path.
	`url="c:\Web\MyApp\Resources\providers.config"`
	`-or-`
	`url="file://c:\Web\MyApp\Resources\providers.config"`

embedded	Specify the providers.config file to be loaded as an embedded resource in an assembly. Syntax for the embedded attribute is '*[extendednamespace.]filename, the name of the assembly which contains the embedded resource*' `embedded="Resources.providers.config, MyApp.Data"`

4.3.3.3. The <settings> Element

There are three default settings used by the framework.. The settings appropriate for one application may not be appropriate for another. The <settings> element lets you configure these options and optimizations for the DataMapper instance that is created from the XML document. Each <settings>

attribute has a default, and you can omit the <settings> element or any of its attributes. The <settings> attributes and the behavior they control are described in the following table.

Table 4.7. Attributes of the <settings> element

Attribute	Description

cacheModelsEnabled	This setting globally enables or disables all cache models for an DataMapper client. This can come in handy for debugging.
	`Example: cacheModelsEnabled=”true”`
	`Default: true (enabled)`

useStatementNamespaces	With this setting enabled, you must always refer to mapped statements by their fully qualified name, which is the combination of the sqlMap namesource and the statement id. For example: `queryForObject(“Namespace.statement.Id”);`
	`Example: useStatementNamespaces=”false”`
	`Default: false (disabled)`

validateSqlMap	This setting globally enables or disables the validation of mapping files against the SqlMapConfig.xsd schema. This can come in handy for debugging.
	`Example: validateSqlMap=”false”`
	`Default: false (disabled)`

useReflectionOptimizer	This setting globally enables or disables the usage of reflection to access property/filed value of C# object. The reflection optimizer use will emit types for retrieving, populating, instantiating parameter and result objects.
	`Example: useReflectionOptimizer=”true”`
	`Default: true (enabled)`

4.3.3.4. The <typeAlias> Element

The <typeAlias> element lets you specify a shorter name in lieu of a fully-qualified classname. For example:

```
<typeAlias alias="LineItem" type="NPetshop.Domain.Billing.LineItem, NPetshop.Domain" />
```

You can then refer to `LineItem` where you would normally have to spell-out the fully qualified class name.

Note

In the .NET implementation, zero or more <typeAlias> elements can appear in the Data Map definition file, within an enclosing <alias> element.

4.3.3.4.1. <typeAlias> attributes

The <typeAlias> element has two attributes:

Table 4.8. Attributes of the <typeAlias> element

Attribute	Description
alias	A unique identifier for this element `alias="Category"`
type	The fully-qualified classname, including namespace reference `type= "IBatisNet.Test.Domain.Category, IBatisNet.Test"`

4.3.3.4.1.1. type Attribute

When specifying a type attribute for the configuration, the type attribute value must be a fully qualified type name in the following format:

```
type="[namespace.class], [assembly name],
     Version=[version], Culture=[culture],
     PublicKeyToken=[public token]"
```

For example:

```
type="MyProject.Domain.LineItem, MyProject.Domain,
     Version=1.2.3300.0, Culture=neutral,
     PublicKeyToken=b03f455f11d50a3a"
```

The strongly typed name is desired, although it is also legitimate to use the shorter style assembly type name:

```
type="MyProject.Domain.LineItem, MyProject.Domain"
```

4.3.3.4.2. Predefined type aliases

The framework predefines some aliases that you can use in your DataMapper configuration and data map files, as shown in Table 4.9.

Table 4.9. Predefined Aliases

CLR Type	Alias
System.ArrayList	list
System.Boolean	Boolean, bool

CLR Type	Alias
System.Byte	Byte, byte
System.Char	Char, char
System.DateTime	dateTime, date
System.Decimal	Decimal, decimal
System.Double	Double, double
System.Guid	guid
System.Hashtable	map, hashmap, hashtable
System.Int16	Int16, short, Short
System.Int32	Int32, int, Int, integer, Integer
System.Int64	Int64, long, Long
System.SByte	SByte, sbyte
System.Single	Float, float, Single, single
System.String	String, string
System.TimeSpan	N/A
System.UInt16	Short, short
System.UInt32	Uint, uint
System.UInt64	Ulong, ulong

4.3.3.5. The <typeHandler> Element

The <typeHandler> element allows for the configuration and use of a Custom Type Handler (see the Custom Type Handler section). This extends the DataMapper's capabilities in handling types that are specific to your database provider, are not handled by your database provider, or just happen to be a part of your application design.

```
<typeHandler type="guid" dbType="Varchar2" callback="GuidVarchar"/>
```

Note

The DataMapper for .NET allows for zero or more <typeAlias> elements to appear in the Data Map definition file, within an enclosing <alias> element.

4.3.3.5.1. <typeHandler> attributes

The <typeHandler> element has three attributes:

Table 4.10. Attributes of the <typeAlias> element

Attribute	Description
type	Refers to the name of the type to handle `type="guid"`

Attribute	Description
dbType	Indicates the provider dbType to handle `dbType="Varchar2"` Note: Omit this attribute if you want the type handler to replace the default iBATIS type handler.
callback	The alias of the custom type handler class name `callback="GuidVarchar"`

4.3.3.6. The <database> Element

The <database> element encloses elements that configure the database system for use by the framework. These database configuration elements are the <provider> and <datasource> elements.

4.3.3.6.1. The <provider> Element

If the default provider is being used, the <provider> element is optional. Or, if several providers are available, one may be selected using the provider element without modifying the `providers.config` file.

```
<provider name="OleDb1.1"  />
```

4.3.3.6.2. The <datasource> element

The <datasource> element specifies the ADO.NET connection string. Example 4.2, shows sample elements for SQL Server, Oracle, Access, MySql, and PostgreSQL.

Example 4.2. Sample <datasource> and <provider> elements (.NET)

```
<!-- The ${properties} are defined in an external file, -->
<!-- but the values could also be coded inline. -->

<!-- Connecting to SQL Server -->
<database>
  <provider name="sqlServer1.1" />
  <dataSource name="NPetstore" default="true"
       connectionString="data source=(local)\NetSDK;database=${database};
       user id=${username};password=${password};"/>
</database>

<!-- Connecting to Oracle -->
<database>
  <provider name="oracleClient1.0"/>
  <dataSource name="iBatisNet"
       connectionString="Data Source=${datasource};User Id=${userid};Password=${password}"/>
</database>

<!-- Connecting to Access -->
<database>
  <provider name="OleDb1.1" />
  <dataSource name="NPetstore" default="true"
       connectionString="Provider=Microsoft.Jet.OLEDB.4.0;Data Source=${database}"/>
</database>

<!-- Connecting to a MySQL database -->
<database>
```

```
  <provider name="ByteFx" />
  <dataSource name="NPetstore" default="true"
       connectionString="Host=${host};Database=${database};
       Password=${password};Username=${username}" />
</database>

<!-- Connecting to a PostgreSQL database -->
<database>
  <provider name="PostgreSql0.7" />
  <dataSource name="NPetstore" default="true"
       connectionString="Server=${server};Port=5432;User Id=${userid};Password=${password};
       Database=${database};" />
</database>
```

4.3.3.7. The <sqlMap> Element

On a daily basis, most of your work will be with the Data Maps, which are covered by Section 3. The Data Maps define the actual SQL statements or stored procedures used by your application. The parameter and result objects are also defined as part of the Data Map. As your application grows, you may have several varieties of Data Map. To help you keep your Data Maps organized, you can create any number of Data Map definition files and incorporate them by reference in the DataMapper configuration. All of the definition files used by a DataMapper instance must be listed in the configuration file.

Example 4.3 shows <sqlMap> elements for loading a set of Data Map definitions. Note that the <sqlMap> elements are nested in a <sqlMaps> element. For more about Data Map definition files, see Section 3.

Example 4.3. Specifying sqlMap locations

```
<!-- Relative path from the project root directory using a property variable -->
<sqlMaps>
  <sqlMap resource="${root}Maps/Account.xml"/>
  <sqlMap resource="${root}Maps/Category.xml"/>
  <sqlMap resource="${root}Maps/Product.xml"/>
</sqlMaps>

<!-- Embedded resources using [extendednamespace.]filename, assemblyname -->
<sqlMaps>
  <sqlMap embedded="Maps.Account.xml, MyApp.Data"/>
  <sqlMap embedded="Maps.Category.xml, MyApp.Data"/>
  <sqlMap embedded="Maps.Product.xml, MyApp.Data"/>
</sqlMaps>

<!-- Full URL with a property variable -->
<sqlMaps>
  <sqlMap url="C:/${projectdir}/MyApp/Maps/Account.xml"/>
  <sqlMap url="C:/${projectdir}/MyApp/Maps/Category.xml"/>
  <sqlMap url="C:/${projectdir}/MyApp/Maps/Product.xml"/>
</sqlMaps>
```

Tip

Since the application root directory location differs by project type (Windows, Web, or library), it is best to use a properties variable to indicate the relative path when using the <sqlMap> "resource" attribute. Having a variable defined in a properties file makes it easy to change the path to all your Data Mapper configuration resources in one location (note the ${projectdir} and ${root} variables in the example above).

4.4. Programming with iBATIS DataMapper: The .NET API

The IBATIS.NET DataMapper API provides four core functions:

1. build a `SqlMapper` instance from a configuration file
2. execute an update query (including insert and delete).
3. execute a select query for a single object
4. execute a select query for a list of objects

The API also provides support for retrieving paginated lists and managing transactions.

4.4.1. Building a SqlMapper Instance

Important

In prior versions of the DataMapper, the `SqlMapper` class was responsible for configuration. This has been superceded by a new configuration API found within the `DomSqlMapBuilder` class. Old configuration method signatures have remained the same, but there are new methods that have been added for more flexibility. These methods support the loading of configuration information through a `Stream`, `Uri`, `FileInfo`, or `XmlDocument` instance.

An XML document is a wonderful tool for describing a database configuration (Section 4.3) or defining a set of data mappings (Section 3), but you can't *execute* XML. In order to use the iBATIS.NET configuration and definitions in your .NET application, you need a class you can call.

The framework provides service methods that you can call which read the configuration file (and any of its definition files) and builds a `SqlMapper` object. The `SqlMapper` object provides access to the rest of the framework. The `SqlMapper` is designed to be multi-threaded and long-lived, and so makes for a good singleton. Example 76 shows a singleton Mapper that is bundled with the framework.

Example 4.4. A Mapper singleton you can call from your own applications

```
[C#]
using IBatisNet.Common.Utilities;
using IBatisNet.DataMapper;
using IBatisNet.DataMapper.Configuration;

namespace IBatisNet.DataMapper
{
 public class Mapper
 {
  private static volatile ISqlMapper _mapper = null;

  protected static void Configure (object obj)
  {
   _mapper = null;
  }

  protected static void InitMapper()
  {
   ConfigureHandler handler = new ConfigureHandler(Configure);
   DomSqlMapBuilder builder = new DomSqlMapBuilder();
   _mapper = builder.ConfigureAndWatch(handler);
  }
```

```
  public static ISqlMapper Instance()
  {
   if (_mapper == null)
   {
    lock (typeof (SqlMapper))
    {
     if (_mapper == null) // double-check
     {
      InitMapper();
     }
    }
   }
   return _mapper;
  }

  public static ISqlMapper Get()
  {
   return Instance();
  }
 }
}
```

To obtain the `ISqlMapper` instance, just call

```
[C#]
ISqlMapper mapper = Mapper.Instance();
```

anywhere in your application, and specify one of the `SqlMapper` methods (see Section 5.3.2) . Here's an example:

```
[C#]
IList list = Mapper.Instance().QueryForList("PermitNoForYearList", values);
```

The first time `Mapper.Instance()` is called, the `DomSqlMapBuilder` object will look for the `SqlMap.config` file in the default location for the type of project it is being used in and build a SqlMapper instance from that configuration. On subsequent calls, the cached `mapper` instance will be reused. The `DomSqlMapBuilder.ConfigureAndWatch`() method monitors changes to the configuration files. If the configuration or definitions files change, the `SqlMapper` will be safely reloaded. This is particularly useful in development, when you might make a change to a data map definition and want to see it take effect without restarting a debugging session. Likewise, in production, it can allow you to make changes to the definitions without reloading the rest of the application.

Tip

If you are using NUnit to test your mappings, you can run a test suite, make changes to the XML mapping document, and run the test again. NUnit will reload the configuration automatically.

Note

The ConfigureAndWatch method requires that your SqlMap.config file and data map files are accessible through the application's file system to be able to track file changes.

If for some reason you do not want to monitor changes to the configuration, you can create your own `Mapper` class, and use the `Configure` method instead:

```
[C#]
ISqlMapper mapper = builder.Configure();
```

4.4.1.1. Multiple Databases

If you need access to more than one database from the same application, create a DataMapper configuration file for that database and another `Mapper` class to go with it. In the new `Mapper` class, change the call to `ConfigureAndWatch` to

```
[C#]
ISqlMapper mapper = builder.ConfigureAndWatch("anotherSqlMapConfig.config", handler);
```

and substitute the name of your configuration file. Each database then has their own singleton you can call from your application:

```
[C#]
ISqlMapper sqlServer = SqlServerMapper.Get();
ISqlMapper access = AccessMapper.Get();
```

4.4.1.2. DomSqlMapBuilder Configuration Options

iBATIS offers you a plethora of other options for loading your `SqlMap.config` file such as loading it through a `Stream`, `Uri`, `FileInfo`, or `XmlDocument` instance. All of these methods are available through the `DomSqlMapBuilder` API for creating a `SqlMapper` instance.

As seen in the prior section, the basic `DomSqlMapBuilder.Configure()` call will look for a file named `SqlMap.config` in your application's root directory. This directory's location differs by project type but is normally the directory where you place your web.config or app.config file.

Example 4.5. Basic SqlMapper Configuration Call

```
ISqlMapper mapper = builder.Configure();
```

If you have named your configuration file something other than SqlMap.config or if you have located your configuration file in a directory other than the application root directory, you can also pass in a relative or absolute file path to the `Configure` method.

Example 4.6. SqlMapper Configuration through an absolute or relative file path

```
/* Configure a SqlMapper from a file path.
   Uses a relative resource path from your application root
   or an absolute file path such as "file:\\c:\dir\a.config" */
ISqlMapper mapper = builder.Configure(strPath);
```

Tip

Since the application root directory location differs by project type (Windows, Web, or library), you can use an AppSettings key for defining a relative path to your SqlMap.config file. Having this key defined makes it easy to change the path without having to recompile your code:

```
mapper = builder.Configure(
         ConfigurationSettings.AppSettings["rootPath"]+"SqlMap.config");
```

Aside from using a simple string filepath, you can also pass in a `FileInfo` or `Uri` instance for the `DomSqlMapBuilder` to use in locating your `SqlMap.config` file.

Example 4.7. SqlMapper Configuration with a FileInfo or Uri instance

```
/* Configure a SqlMapper with FileInfo. */
FileInfo aFileInfo = someSupportClass.GetDynamicFileInfo();
ISqlMapper mapper = builder.Configure(aFileInfo);

/* Configure a SqlMapper through a Uri. */
Uri aUri = someSupportClass.GetDynamicUri();
ISqlMapper anotherMapper = builder.Configure(aUri);
```

If you find that you already have loaded your DataMapper configuration information as an `XmlDocument` or `Stream` instance within your application, the `DomSqlMapBuilder` provides `Configure` overloads for those types as well.

Example 4.8. SqlMapper Configuration with an XmlDocument or Stream

```
/* Configure a SqlMapper with an XmlDocument */
XmlDocument anXmlDoc = someSupportClass.GetDynamicXmlDocument();
ISqlMapper mapper = builder.Configure(anXmlDoc);

/* Configure a SqlMapper from a stream. */
Stream aStream = someSupportClass.GetDynamicStream();
ISqlMapper anotherMapper = builder.Configure(aStream);
```

The `DomSqlMapBuilder` API provides `ConfigureAndWatch` methods that can be used to monitor changes to the configuration files. This is particularly useful when using a singleton such as the `Mapper` class shown in the prior section. The example `Mapper` singleton allows a reconfigured `SqlMapper` instance to be reloaded on the fly.

Example 4.9. Mapper ConfigureHandler delegate

```
...
  protected static void Configure (object obj)
  {
   _mapper = null;
  }

  protected static void InitMapper()
  {
   ConfigureHandler handler = new ConfigureHandler(Configure);
   DomSqlMapBuilder builder = new DomSqlMapBuilder();
   _mapper = builder.ConfigureAndWatch(handler);
  }
...
```

If you use a custom singleton, you will need to pass a `ConfigureHandler` (callback delegate) to the `DomSqlMapBuilder` so that it knows the method for resetting your application's `SqlMapper` instance. In the `Mapper`'s case, its `Configure` method is used as the callback delegate.

Since the configuration files need to be watched for changes, your SqlMap.config file must be accessible through the file system. This means that configuration is limited to the three methods

shown below.

Example 4.10. DomSqlMapBuilder ConfigureAndWatch methods

```
/* Configure and monitor the configuration file for modifications
   and automatically reconfigure the SqlMapper.
   This basic ConfigureAndWatch method looks for a file with the
   default name of SqlMap.config in the application root directory. */
public ISqlMapper ConfigureAndWatch(ConfigureHandler configureDelegate)

/* Configure and monitor the configuration file for modifications
   and automatically reconfigure the SqlMapper.
   Uses a relative path from your application root
   or an absolute file path such as "file:\\c:\dir\a.config" */
public ISqlMapper ConfigureAndWatch( string resource, ConfigureHandler configureDelegate )

/* Configure and monitor the configuration file for modifications
   and automatically reconfigure the SqlMapper.
   Uses a FileInfo instance for your config file. */
public ISqlMapper ConfigureAndWatch( FileInfo resource, ConfigureHandler configureDelegate )
```

4.4.1.3. DomSqlMapBuilder : Advanced settings

Before launching the 'Configure' method to build the ISqlMapper instance, you can set those porperties.

Table 4.11. Advanced settings

Propertie	Description

Properties	Allow to set properties before configuration. Those properties will be added to the properties list defined in the properties.config.
	`NameValueCollection properties = new NameValueCollection();`
	`properties.Add("connectionString", "...");`
	`builder.Properties = properties;`
	
	`ISqlMapper mapper = builder.Configure("sqlMap.config");`

GetAccessorFactory	Allow to set a custom get accessor factory before configuration, see IGetAccessorFactory interface which defines the contract for the factory responsible to build set accessor for a member object in iBATIS.
SetAccessorFactory	Allow to set a custom set accessor factory before configuration, see ISetAccessorFactory interface which defines the contract for the factory responsible to build get accessor for a member object in iBATIS.
ObjectFactory	Allow to set a custom object factory before configuration, see IObjectFactory interface which defines the contract for the factory responsible for object creation in iBATIS.
ValidateSqlMapConfig	Enable whether or not the validation of configuration document before configuration

4.4.2. Exploring the DataMapper API through the SqlMapper

The `SqlMapper` instance acts as a facade to provide access the rest of the DataMapper framework. The DataMapper API methods are shown in Example 4.11.

Example 4.11. The DataMapper API for .NET

```
[C#]

/* Query API */
public object Insert(string statementName, object parameterObject);
public int Update(string statementName, object parameterObject);
public int Delete(string statementName, object parameterObject);

public object QueryForObject(string statementName, object parameterObject);
public T QueryForObject<T>(string statementName, object parameterObject);
public object QueryForObject(string statementName, object parameterObject, object resultObject);
public T QueryForObject<T>(string statementName, object parameterObject, T resultObject);

public IList QueryForList(string statementName, object parameterObject);
public IList<T> QueryForList<T>(string statementName, object parameterObject);
public void QueryForList(string statementName, object parameterObject, IList resultObject);
public void QueryForList<T>(string statementName, object parameterObject, IList<T> resultObject);
public IList QueryForList(string statementName, object parameterObject,
                          int skipResults, int maxResults);
public IList<T> QueryForList<T>(string statementName, object parameterObject,
                          int skipResults, int maxResults);

public IList QueryWithRowDelegate(string statementName, object parameterObject,
                                  RowDelegate rowDelegate);
public IList<T> QueryWithRowDelegate<T>(string statementName, object parameterObject,
                                 SqlMapper.RowDelegate<T> rowDelegate);

public PaginatedList QueryForPaginatedList(String statementName, object parameterObject,
                                           int pageSize);

public IDictionary QueryForDictionary(string statementName, object parameterObject,
                                      string keyProperty)
public IDictionary QueryForDictionary(string statementName, object parameterObject,
                                      string keyProperty, string valueProperty)
public IDictionary QueryForMap(string statementName, object parameterObject, string keyProperty)
public IDictionary QueryForMap(string statementName, object parameterObject,
                               string keyProperty, string valueProperty)

/* Connection API */
public void OpenConnection()
public void CloseConnection()

/* Transaction API */
public void BeginTransaction()
public void BeginTransaction(bool openConnection)
public void BeginTransaction(IsolationLevel isolationLevel)
public void BeginTransaction(bool openConnection, IsolationLevel isolationLevel)

public void CommitTransaction()
public void CommitTransaction(bool closeConnection)

public void RollBackTransaction()
public void RollBackTransaction(bool closeConnection)
```

Note that each of the API methods accept the name of the Mapped Statement as the first parameter. The `statementName` parameter corresponds to the *`id`* of the Mapped Statement in the Data Map definition (see Section 3.3). In each case, a *`parameterObject`* also may be passed. If the Mapped Statement expects no parameters, a null *`parameterObject`* may be passed. If a statement does expect parameters, then a valid *`parameterObject`* is required. The following sections describe how the API methods work.

4.4.2.1. Insert, Update, Delete

```
public object Insert(string statementName,
                     object parameterObject);
public int Update(string statementName,
                  object parameterObject);
public int Delete(string statementName,
                  object parameterObject);
```

If a Mapped Statement uses one of the <insert>, <update>, or <delete> statement-types, then it should use the corresponding API method. The <insert> element supports a nested <selectKey> element for generating primary keys (see Section 3.3.3). If the <selectKey> stanza is used, then `Insert` returns the generated key; otherwise a null object is returned. Both the `Update` and `Delete` methods return the number of rows affected by the statement.

4.4.2.2. QueryForObject

```
public object QueryForObject(string statementName,
                             object parameterObject);
public object QueryForObject(string statementName,
                             object parameterObject,
                             object resultObject);

public T QueryForObject<T>(string statementName, object parameterObject);
public T QueryForObject<T>(string statementName, object parameterObject, T resultObject);
```

If a Mapped Statement is expected to select a single row, then call it using `QueryForObject`. Since the Mapped Statement definition specifies the result class expected, the framework can both create and populate the result class for you. Alternatively, if you need to manage the result object yourself, say because it is being populated by more than one statement, you can use the alternate form and pass your *`resultObject`* as the third parameter.

4.4.2.3. QueryForList

```
public IList QueryForList(string statementName,
                          object parameterObject);
public void QueryForList(string statementName,
                         object parameterObject,
                         IList resultObject);
public IList QueryForList(string statementName,
                          object parameterObject,
                          int skipResults,
                          int maxResults);

public IList<T> QueryForList<T>(string statementName, object parameterObject);
public void QueryForList<T>(string statementName,
                                  object parameterObject,
                                                         IList<T> resultObject);
public IList<T> QueryForList<T>(string statementName,
                                        object parameterObject,
                                        int skipResults, int maxResults);
```

If a Mapped Statement is expected to select multiple rows, then call it using `QueryForList` . Each entry in the list will be an result object populated from the corresponding row of the query result. If you need to manage the resultObject yourself, then it can be passed as the third parameter.

If you need to obtain a partial result, the third form takes the number of records to skip (the starting point) and the maximum number to return, as the *skipResults* and *maxResults* parameters. The PaginatedList method provides the same functionality but in a more convenient wrapper. The QueryWithRowDelegate method also works with multiple rows, but provides a post-processing feature.

4.4.2.4. QueryWithRowDelegate

```
public delegate void RowDelegate(object obj,
                                 IList list);

public IList QueryWithRowDelegate(string statementName,
                                  object parameterObject,
                                  RowDelegate rowDelegate);

public IList<T> QueryWithRowDelegate<T>(string statementName, object parameterObject,
                                SqlMapper.RowDelegate<T> rowDelegate);
```

No matter how well our database is designed or how cleverly we describe our maps, the result objects we get back may not be ideal. You may need to perform some post-processing task on the result objects. You might even want to omit an entry omitted from the list. Or, you might want to use the result object to create some other, more useful object. To save filtering the result objects from to one list to another, you can pass a RowDelegate to the method to do the dirty work. The SqlMapper will go through each of the result objects and give the delegate a chance to modify the object and determine if the object should be added to the IList that will be returned.

Important

It is your responsibility to add the objects you want returned to the list. If an object is not added, it is not returned.

4.4.2.5. QueryForMapWithRowDelegate

```
public delegate void DictionaryRowDelegate(object key,
                                           object value,
                                           object parameterObject,
                                           IDictionary dictionary);

public IDictionary QueryForMapWithRowDelegate(string statementName,
                                              object parameterObject,
                                              string keyProperty,
                                              string valueProperty,
                                              DictionaryRowDelegate rowDelegate);
```

No matter how well our database is designed or how cleverly we describe our maps, the result objects we get back may not be ideal. You may need to perform some post-processing task on the result objects. You might even want to omit an entry omitted from the dictionary. Or, you might want to use the result object to create some other, more useful object. To save filtering the result objects from to one dictionary to another, you can pass a DictionaryRowDelegate to the method to do the dirty work. The SqlMapper will go through each of the result objects and give the delegate a chance to modify the object and determine if the object should be added to the IDictionary that will be returned.

Important

It is your responsibility to add the objects you want returned to the dictionary. If an object is not

added, it is not returned.

4.4.2.6. QueryForPaginatedList

```
public PaginatedList QueryForPaginatedList(string statementName,
                                           object parameterObject,
                                           int pageSize);
```

We live in an age of information overflow. A database query often returns more hits than users want to see at once, and our requirements may say that we need to offer a long list of results a "page" at a time. If the query returns 1000 hits, we might need to present the hits to the user in sets of fifty, and let them move back and forth between the sets. Since this is such a common requirement, the framework provides a convenience method.

The `PaginatedList` interface includes methods for navigating through pages (`nextPage()`, `previousPage()`, `gotoPage()`) and also checking the status of the page (`isFirstPage()`, `isMiddlePage()`, `isLastPage()`, `isNextPageAvailable()`, `isPreviousPageAvailable()`, `getPageIndex()`, `getPageSize()`). Although the total number of records available is not accessible from the `PaginatedList` interface, this should be easily accomplished by simply executing a second statement that counts the expected results. Too much overhead would be associated with the `PaginatedList` otherwise.

Tip

The `PaginatedList` method is convenient, but note that a larger set will first be returned by the database provider and the smaller set extracted by the framework. The higher the page, the larger set that will be returned and thrown away. For very large sets, you may want to use a stored procedure or your own query that uses *`skipResults`* and *`maxResults`* as parameters. Unfortunately, the semantics for the returning partial data sets is not standardized, so `PaginatedList` is the best we can do within the scope of a framework.

4.4.2.7. QueryForDictionary, QueryForMap

```
public IDictionary QueryForDictionary(string statementName,
                                      object parameterObject,
                                      string keyProperty)
public IDictionary QueryForDictionary(string statementName,
                                      object parameterObject,
                                      string keyProperty,
                                      string valueProperty)
public IDictionary QueryForMap(string statementName,
                               object parameterObject,
                               string keyProperty)
public IDictionary QueryForMap(string statementName,
                               object parameterObject,
                               string keyProperty,
                               string valueProperty)
```

The `QueryForList` methods return the result objects within a `IList` instance. Alternatively, the `QueryForDictionary` returns a `IDictionary` instance. The value of each entry is one of the result objects. The key to each entry is indicated by the *`keyProperty`* parameter. This is the name of the one of the properties of the result object, the value of which is used as the key for each entry. For example, If you needed a set of `Employee` objects, you might want them returned as a `IDictionary` keyed by each object's EmployeeNumber property.

If you don't need the entire result object in your `Dictionary`, you can add the *`valueProperty`*

parameter to indicate which result object property should be the value of an entry. For example, you might just want the EmployeeName keyed by EmployeeNumber.

Important

You do not need to use this method just to obtain an `IDictionary` result object. As explained in Section 3.5, the result object for any query can be a property object or a `IDictionary` instance. This method is used to generate a *new* `IDictionary` result object from a property object or (another) `IDictionary` object. In this case, the key is a property you specify, and the value is the row from the result set.

The `QueryforMap` methods provide the same functionality but under a different name, for the sake of consistency with the Java implementation. (The .NET `IDictionary` interface is equivalent to the Java `Map` interface.)

4.4.3. Session

In the iBATIS DataMapper framework, a session is a container for an ADO connection and transaction.

The DataMapper's `IDalSession` implements the `IDisposable` interface. So you can use it with the `using` syntax.

Example 4.12. using instruction

```
[C#]
using ( IDalSession session = sqlMap.OpenConnection() )
{
  Account account = sqlMap.QueryForObject("GetAccountViaColumnName", 1) as Account;
}
```

Note

Sessions cannot be nested. An exception will be thrown if you call `BeginTransaction`/`OpenConnection` from the same thread more than once or call `CommitTransaction` or `RollbackTransaction` first. In other words, each thread can have *at most* one session open, per `SqlMapper` instance.

4.4.4. Connection

The DataMapper API includes methods to demarcate connection boundaries.

```
// Open a session : Open an ADO connection
public void OpenConnection()
// Close a session : Close the associated ADO connection
public void CloseConnection()
```

Example 4.13. Connection example

```
[C#]
sqlMap.OpenConnection()
Account account = sqlMap.QueryForObject("GetAccountViaColumnName", 1) as Account;
sqlMap.CloseConnection()
```

```
// Same thing with using instruction
using ( IDalSession session = sqlMap.OpenConnection() )
{
  Account account = sqlMap.QueryForObject("GetAccountViaColumnName", 1) as Account;
}
```

4.4.5. Automatic Session

By default, calling any of the API methods (see Section 4.4.2) on a `SqlMapper` instance will auto-open/close a connection. This means that each call to these methods will be a single unit of work. For many cases, this simple approach may be sufficient. But it is not ideal if you have a number of statements that must execute as a single unit of work, which is to say, succeed or fail as a group. For cases like these, you can use *explicit transactions*.

An example of using automatic session is shown as Example 4.14.

Example 4.14. Using automatic session

```
[C#]
Item item = (Item) sqlMap.executeQueryForObject ("GetItem", itemId);
item.Description = “TX1”;
// No session demarcated, so open/close connection will be automatic (implied)
sqlMap.Update("UpdateItem", item);
item.Description = newDescription;
item.Description = “TX2”;
// No transaction demarcated, so open/close connection will be automatic (implied)
sqlMap.Update("UpdateItem", item);
```

Note

Be careful to consider sessions when framing your queries. Automatic sessions are convenient, but you will run into trouble if your unit of work requires more than a single update to the database. In Example 4.14, if the second call to *`"UpdateItem"`* fails, the item description will still be updated with the first new description of *`“TX1”`*. Not what a user might expect.

4.4.6. Transaction

The DataMapper API includes methods to demarcate transactional boundaries. A transaction can be started, committed and/or rolled back. You can call the transaction methods from the `SqlMapper` instance.

```
// Begin a transactional session : Open a connection and begin an ADO transaction
public void BeginTransaction()
// Begin a transactional session : Open a connection is specified and begin an ADO transaction
public void BeginTransaction(bool openConnection)
// Begin a transactional session : Open a connection and begin an ADO transaction
// with the specified IsolationLevel
public void BeginTransaction(IsolationLevel isolationLevel)
// Begin a transactional session : Open a connection is specified and begin an ADO transaction
// with the specified IsolationLevel
public void BeginTransaction(bool openConnection, IsolationLevel isolationLevel)


// Commit a session : Commit the ADO transaction and close the connection
public void CommitTransaction()
// Commit a session : Commit the ADO transaction and close the connection if specified
```

```
public void CommitTransaction(bool closeConnection)
// RollBack a session : RollBack the ADO transaction and close the connection
public void RollBackTransaction()
// RollBack a session : RollBack the ADO transaction and close the connection if specified
public void RollBackTransaction(bool closeConnection)
```

An example of using transactions is shown as Example 4.15.

Example 4.15. Using transactions

```
[C#]
try
  {
      sqlMap.BeginTransaction();
      Item item = (Item) sqlMap.QueryForObject("getItem", itemId);
      item.Description = newDescription;
      sqlMap.Update("updateItem", item);
      sqlMap.CommitTransaction();
  }
catch {
      sqlMap.RollBackTransaction();
}

// With "using" syntax
using ( IDalSession session = sqlMap.BeginTransaction() )
{
  Item item = (Item) sqlMap.QueryForObject("getItem", itemId);
  item.Description = newDescription;
  sqlMap.Update("updateItem", item);

  session.Complete(); // Commit
}
```

4.4.7. Distributed Transactions

Distributed transactions are transactions that can span multiple resource managers, such as SQL Server and Oracle, and reconcile transactions among them.

iBATIS.NET introduces a new `TransactionScope` class mimicking the new `TransactionScope` found in the `System.Transactions` namespace (.NET Framework 2.0). This class supports MSMQ, ADO.NET, SqlServer, and DTC transaction models. This is a simple managed interface to COM+'s SWC (Services Without Components) Transactions. It can be used only by developers using .NET 1.1 and Windows XP SP2 or Windows Server 2003 since it implements distributed transactional support using the `ServiceDomain` class.

Usage is simple, as seen in the following example where a code block is made transactional à la Indigo (moving to Indigo will be easier since it is the same API):

Example 4.16. Using distributed transactions

```
[C#]
using IBatisNet.Common.Transaction;

using (TransactionScope tx = new TransactionScope())
{
   sqlMapSqlServer.OpenConnection();
    // Transaction will be automatically associated
   account = sqlMapSqlServer.QueryForObject("GetAccount", accountId) as Account;
```

```
   account.FirstName = "Gilles";
   sqlMapSqlServer.Update(account);
   sqlMapSqlServer.CloseConnection();

   sqlMapOracle.OpenConnection();
   // Transaction will be automatically associated
   product = sqlMapOracle.QueryForObject("GetProduct", productId) as Product;
   product.Quantity = 1000;
   sqlMapOracle.Update(product);
   sqlMapOracle.CloseConnection();

   tx.Complete(); // Commit
}
```

It is important to make sure that each instance of this class gets `Close()`'d. The easiest way to ensure that each instance is closed is with the `using` statement in C#. When `Dispose` is called on the transaction scope at the end of the `using` code block, the *ambient* transaction will be commited only if the `Complete()` method has been called.

Note

This `TransactionScope` class does not support a nested transaction scope with different transaction options.

4.4.8. Coding Examples [TODO: Expand in to a Cookbook of practical examples]

Example 4.17. Executing Update (insert, update, delete)

```
[C#]
Product product = new Product();
product.Id = 1;
product.Description = “Shih Tzu”;
int key = sqlMap.Insert (“insertProduct”, product);
```

Example 4.18. Executing Query for Object (select)

```
[C#]
int key = 1;
Product product = sqlMap.QueryForObject (“getProduct”, key) as Product;
```

Example 4.19. Executing Query for Object (select) With Preallocated Result Object

```
[C#]
Customer customer = new Customer();
sqlMap.BeginTransaction();
sqlMap.QueryForObject(“getCust”, parameterObject, customer);
sqlMap.QueryForObject(“getAddr”, parameterObject, customer);
sqlMap.CommitTransaction();
```

Example 4.20. Executing Query for List (select)

```
[C#]
IList list = sqlMap.QueryForList (“getProductList”, null);
```

Example 4.21. Auto-Open/Close

```
[C#]
// When OpenConnection is not called, the statements will auto-Open/Close.
int key = sqlMap.Insert (“insertProduct”, product);
```

Example 4.22. Executing Query for List (select) With Result Boundaries

```
[C#]
List list = sqlMap.queryForList (“getProductList”, null, 0, 40);
```

Example 4.23. Executing Query with a RowHandler (select)

```
[C#]
public void RowHandler(object obj, IList list)
{
  Product product = (Product) object;
  product.Quantity = 10000;
}

SqlMapper.RowDelegate handler = new SqlMapper.RowDelegate(this.RowHandler);
IList list = sqlMap.QueryWithRowDelegate("getProductList", null, handler);
```

Example 4.24. Executing Query for Paginated List (select)

```
[C#]
PaginatedList list = sqlMap.QueryForPaginatedList (“getProductList”, null, 10);
list.NextPage();
list.PreviousPage();
```

Example 4.25. Executing Query for Map

```
[C#]
IDictionary map = sqlMap.QueryForMap (“getProductList”, null, “productCode”);
Product p = (Product) map[“EST-93”];
```

4.5. Logging SqlMap Activity

The iBATIS DataMapper framework records its interaction with the database through an internal logging mechanism patterned after Apache Log4Net. The internal logging mechanism can use one of the three built-in loggers (NoOpLogger, ConsoleOutLogger, TraceLogger) or external logging packages such as Apache Log4Net. In order for iBATIS to generate log messages, the application's config file (App.Config or Web.Config) must contain an appropriate configSection node:

Example 4.26. iBATIS Configuration Section Handler for logging

```
<configSections>
 <sectionGroup name="iBATIS">
  <section name="logging" type="IBatisNet.Common.Logging.ConfigurationSectionHandler, IBatisNet.Common" />
 </sectionGroup>
</configSections>
```

The application's config file must declare one logger implementation. See the examples below on how to configure one of the three built-in loggers.

```
<iBATIS>
 <logging>
  <logFactoryAdapter type="IBatisNet.Common.Logging.Impl.ConsoleOutLoggerFA, IBatisNet.Common">
   <arg key="showLogName" value="true" />
   <arg key="showDataTime" value="true" />
   <arg key="level" value="ALL" />
   <arg key="dateTimeFormat" value="yyyy/MM/dd HH:mm:ss:SSS" />
  </logFactoryAdapter>
 </logging>
</iBATIS>
```

```
<iBATIS>
 <logging>
  <logFactoryAdapter type="IBatisNet.Common.Logging.Impl.NoOpLoggerFA, IBatisNet.Common" />
 </logging>
</iBATIS>
```

```
<iBATIS>
 <logging>
  <logFactoryAdapter type="IBatisNet.Common.Logging.Impl.TraceLoggerFA, IBatisNet.Common" />
 </logging>
</iBATIS>
```

To configure iBATIS to use another logger implementation, simple specify the appropriate logFactoryAdapter type. To use Apache Log4Net with the iBATIS DataMapper framework, use the following configuration setting:

```
<iBATIS>
 <logging>
  <logFactoryAdapter type="IBatisNet.Common.Logging.Impl.Log4NetLoggerFA, IBatisNet.Common.Logging.Log4Net">
   <arg key="configType" value="inline" />
  </logFactoryAdapter>
 </logging>
</iBATIS>
```

```
<iBATIS>
 <logging>
  <logFactoryAdapter type="IBatisNet.Common.Logging.Impl.Log4NetLoggerFA, IBatisNet.Common.Logging.Log4Net">
   <arg key="configType" value="file" />
   <arg key="configFile" value="log4Net.config" />
  </logFactoryAdapter>
```

```
 </logging>
</iBATIS>
```

The Log4NetLoggerFA supports the following values for the configTypes argument:

Table 4.12. Valid configType values

configType	Description
inline	log4net node will use the log4net node in the App.Config/Web.Config file when it is configured
file	(also requires configFile argument) - log4net will use an external file for its configuration
file-watch	(also requires configFile argument) - log4net will use an external file for its configuration and will re-configure itself if this file changes
external	iBATIS will not attempt to configure log4net.

4.5.1. Sample Logging Configurations

The simplest logging configuration is to output log messages to Console.Out:

```
<configuration>
 <configSections>
  <sectionGroup name="iBATIS">
   <section name="logging" type="IBatisNet.Common.Logging.ConfigurationSectionHandler, IBatisNet.Common" />
  </sectionGroup>
 </configSections>
 <iBATIS>
  <logging>
   <logFactoryAdapter type="IBatisNet.Common.Logging.Impl.ConsoleLoggerFA, IBatisNet.Common.Logging" />
  </logging>
 </iBATIS>
</configuration>
```

A common logging configuration is to use Apache Log4Net. To use Log4Net with your own application, you need to provide your own Log4Net configuration. You can do this by adding a configuration file for your assembly that includes a <log4Net> element. The configuration file is named after your assembly but adds a `.config` extension, and is stored in the same folder as your assembly. This is an example of a basic Log4Net configuration block (IBatisNet.DataMapper.Test.dll.Config) that also creates a log4net.txt which contains debug information from log4net. If log4net is not producing output, check the log4net.txt file.

```
<configuration>
 <configSections>
  <sectionGroup name="iBATIS">
   <section name="logging" type="IBatisNet.Common.Logging.ConfigurationSectionHandler, IBatisNet.Common" />
  </sectionGroup>
  <section name="log4net" type="log4net.Config.Log4NetConfigurationSectionHandler, log4net" />
 </configSections>
 <appSettings>
  <add key="log4net.Internal.Debug" value="true"/>
 </appSettings>
 <system.diagnostics>
```

```
 <trace autoflush="true">
   <listeners>
    <add name="textWriterTraceListener"
     type="System.Diagnostics.TextWriterTraceListener"
     initializeData="C:\\inetpub\\wwwroot\\log4net.txt" />
   </listeners>
  </trace>
 </system.diagnostics>
 <iBATIS>
  <logging>
   <logFactoryAdapter type="IBatisNet.Common.Logging.Impl.Log4NetLoggerFA, IBatisNet.Common.Logging.Log4Net">
    <arg key="configType" value="inline" />
   </logFactoryAdapter>
  </logging>
 </iBATIS>
 <log4net>
  <appender name="FileAppender" type="log4net.Appender.FileAppender">
   <file value="log.txt" />
   <appendToFile value="true" />
   <layout type="log4net.Layout.SimpleLayout" />
  </appender>
  <root>
   <level value="ALL" />
   <appender-ref ref="FileAppender" />
  </root>
 </log4net>
</configuration>
```

Example 4.27. A complete Log4Net configuration block (IBatisNet.DataMapper.Test.dll.Config)

```
<configuration>
 <!-- Register a section handler for the log4net section -->
 <configSections>
  <section name="log4net" type="log4net.Config.Log4NetConfigurationSectionHandler, log4net" />
 </configSections>
 <appSettings>
  <!-- To enable internal log4net logging specify the following appSettings key -->
  <!-- <add key="log4net.Internal.Debug" value="true"/> -->
 </appSettings>

 <!-- This section contains the log4net configuration settings -->
 <log4net>
  <!-- Define some output appenders -->
  <appender name="RollingLogFileAppender" type="log4net.Appender.RollingFileAppender">
   <param name="File" value="log.txt" />
   <param name="AppendToFile" value="true" />
   <param name="MaxSizeRollBackups" value="2" />
   <param name="MaximumFileSize" value="100KB" />
   <param name="RollingStyle" value="Size" />
   <param name="StaticLogFileName" value="true" />
   <layout type="log4net.Layout.PatternLayout">
    <param name="Header" value="[Header]\r\n" />
    <param name="Footer" value="[Footer]\r\n" />
    <param name="ConversionPattern" value="%d [%t] %-5p %c [%x] - %m%n" />
   </layout>
  </appender>
  <appender name="ConsoleAppender" type="log4net.Appender.ConsoleAppender">
   <layout type="log4net.Layout.PatternLayout">
    <param name="ConversionPattern" value="%d [%t] %-5p %c [%x] &lt;%X{auth}&gt; - %m%n" />
   </layout>
  </appender>

  <!-- OFF, FATAL, ERROR, WARN, INFO, DEBUG, ALL -->
  <!-- Set root logger level to ERROR and its appenders -->
  <root>
   <level value="ERROR" />
   <appender-ref ref="RollingLogFileAppender" />
   <appender-ref ref="ConsoleAppender" />
  </root>

  <!-- Print only messages of level DEBUG or above in the packages -->
  <logger name="IBatisNet.DataMapper.Commands.DefaultPreparedCommand">
```

```
   <level value="DEBUG" />
  </logger>
  <logger name="IBatisNet.DataMapper.Configuration.Cache.CacheModel">
   <level value="DEBUG" />
  </logger>
  <logger name="IBatisNet.DataMapper.LazyLoadList">
   <level value="DEBUG" />
  </logger>
  <logger name="IBatisNet.DataMapper.SqlMapSession">
   <level value="DEBUG" />
  </logger>
  <logger name="IBatisNet.Common.Transaction.TransactionScope">
   <level value="DEBUG" />
  </logger>
  <logger name="IBatisNet.DataAccess.DaoSession">
   <level value="DEBUG" />
  </logger>
  <logger name="IBatisNet.DataAccess.Configuration.DaoProxy">
   <level value="DEBUG" />
  </logger>
  <logger name="IBatisNet.DataMapper.Configuration.Statements.PreparedStatementFactory">
   <level value="OFF" />
  </logger>
  <logger name="IBatisNet.DataMapper.Commands.IPreparedCommand">
   <level value="OFF" />
  </logger>
 </log4net>
</configuration>
```

To log all Prepared SQL command text, enable the logger for `IBatisNet.DataMapper.Configuration.Statements.DefaultPreparedCommand`. This will display the statement, parameters, and parameter types used by the DataMapper.

Example 4.28. Sample DefaultPreparedCommand logger output

```
2005-06-08 01:39:33 [3872] DEBUG IBatisNet.DataMapper.Commands.DefaultPreparedCommand -
Statement Id: [User.Update] PreparedStatement : [UPDATE [User] SET [DateLastUpdated] = NOW() WHERE [UserId] = ?]
2005-06-08 01:39:43 [3872] DEBUG IBatisNet.DataMapper.Commands.DefaultPreparedCommand -
Statement Id: [User.Update] Parameters: [param0=[UserId,1]]
2005-06-08 01:39:53 [3872] DEBUG IBatisNet.DataMapper.Commands.DefaultPreparedCommand -
Statement Id: [User.Update] Types: [param0=[String, System.Int32]]
```

If your statements are named the same across all sqlMap files, its possible to filter log messages when using Apache Log4Net. The example below only logs Insert, Update, and Delete statements sent to the database. Select or GetMany statements are not logged:

Example 4.29. Sample Apache Log4Net appender node that will log Insert, Update, and Delete statements to a file

```
<appender name="FileAppender" type="log4net.Appender.FileAppender">
 <file value="InsertsUpdatesDeletes.txt" />
 <layout type="log4net.Layout.SimpleLayout" />
 <filter type="log4net.Filter.StringMatchFilter">
  <regExToMatch value="^(?!Statement Id:)|Statement Id: \[\w+?\.(?:Insert|Update|Delete)" />
 </filter>
 <filter type="log4net.Filter.DenyAllFilter" />
</appender>
```

If you would like to log cache usage in your application, enable the logger for `IBatisNet.DataMapper.Configuration.Cache.CacheModel`.

Example 4.30. Sample CacheModel logger output

```
2005-06-08 01:38:34,403 [3648] DEBUG IBatisNet.DataMapper.Configuration.Cache.CacheModel -
Flush cacheModel named Account.account-cache for statement 'UpdateAccountViaParameterMap'
```

To keep track of DataMapper session information, enable the logger for `IBatisNet.DataMapper.SqlMapSession`.

Example 4.31. Sample SqlMapSession logger output

```
2005-06-08 01:39:42,660 [3872] DEBUG IBatisNet.DataMapper.SqlMapSession [] -
  Open Connection "3194"
  to "Microsoft SQL Server 7.0/2000, provider V1.0.5000.0 in framework .NET V1.1".
2005-06-08 01:39:42,660 [3872] DEBUG IBatisNet.DataMapper.Commands.DefaultPreparedCommand [] -
  PreparedStatement : [select * from Orders where Order_ID =  @param0 ]
2005-06-08 01:39:42,660 [3872] DEBUG IBatisNet.DataMapper.Commands.DefaultPreparedCommand [] -
  Parameters: [@param0=[value,1]]
2005-06-08 01:39:42,660 [3872] DEBUG IBatisNet.DataMapper.Commands.DefaultPreparedCommand [] -
  Types: [@param0=[Int32, System.Int32]]
2005-06-08 01:39:42,676 [3872] DEBUG IBatisNet.DataMapper.SqlMapSession [] -
  Close Connection "3194" to
  "Microsoft SQL Server 7.0/2000, provider V1.0.5000.0 in framework .NET V1.1".
```

To get information about lazyload usage by the DataMapper, enable the logger for `IBatisNet.DataMapper.LazyLoadList`.

Example 4.32. Sample LazyLoadList logger output

```
2005-06-08 01:39:42,316 [3872] DEBUG IBatisNet.DataMapper.LazyLoadList [] -
  Proxyfying call to get_Count
2005-06-08 01:39:42,316 [3872] DEBUG IBatisNet.DataMapper.LazyLoadList [] -
  Proxyfying call, query statement GetLineItemsForOrder
2005-06-08 01:39:42,316 [3872] DEBUG IBatisNet.DataMapper.SqlMapSession [] -
  Open Connection "3463" to
  "Microsoft SQL Server 7.0/2000, provider V1.0.5000.0 in framework .NET V1.1".
2005-06-08 01:39:42,316 [3872] DEBUG IBatisNet.DataMapper.Commands.DefaultPreparedCommand [] -
  PreparedStatement : [select LineItem_ID as Id, LineItem_Code as Code,
  LineItem_Quantity as Quantity, LineItem_Price as Price
  from LineItems where Order_ID =  @param0 ]
2005-06-08 01:39:42,316 [3872] DEBUG IBatisNet.DataMapper.Commands.DefaultPreparedCommand [] -
  Parameters: [@param0=[value,1]]
2005-06-08 01:39:42,316 [3872] DEBUG IBatisNet.DataMapper.Commands.DefaultPreparedCommand [] -
  Types: [@param0=[Int32, System.Int32]]
2005-06-08 01:39:42,316 [3872] DEBUG IBatisNet.DataMapper.SqlMapSession [] -
  Close Connection "3463" to
  "Microsoft SQL Server 7.0/2000, provider V1.0.5000.0 in framework .NET V1.1".
2005-06-08 01:39:42,316 [3872] DEBUG IBatisNet.DataMapper.LazyLoadList [] -
  End of proxyfied call to get_Count
```

Appendix A. iBATIS.NET's

SqlMapConfig.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://ibatis.apache.org/dataMapper"
elementFormDefault="qualified"
xmlns:mstns="http://tempuri.org/XMLSchema.xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="http://ibatis.apache.org/dataMapper"
xmlns:vs="http://schemas.microsoft.com/Visual-Studio-Intellisense"
vs:friendlyname="iBATIS.NET DataMapper Configuration Schema"
vs:ishtmlschema="false"
vs:iscasesensitive="true"
vs:requireattributequotes="true"
vs:defaultnamespacequalifier=""
vs:defaultnsprefix="">

        <xs:annotation>
                <xs:documentation>
                DataMapper XML Schema Definition
        </xs:documentation>
        </xs:annotation>
        <xs:element name="typeAlias">
                <xs:complexType>
                        <xs:attribute name="alias" type="xs:string" use="required"/>
                        <xs:attribute name="type" type="xs:string" use="required"/>
                </xs:complexType>
        </xs:element>
        <xs:element name="typeHandler">
                <xs:complexType>
                        <xs:attribute name="type" type="xs:string"/>
                        <xs:attribute name="dbType" type="xs:string"/>
                        <xs:attribute name="callback" type="xs:string"/>
                </xs:complexType>
        </xs:element>
        <xs:element name="typeHandlers">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="typeHandler" maxOccurs="unbounded"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:element name="alias">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="typeAlias" maxOccurs="unbounded"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:element name="dataSource">
                <xs:complexType>
                        <xs:attribute name="name" type="xs:string" use="required"/>
                        <xs:attribute name="connectionString" type="xs:string" use="required"/>
                </xs:complexType>
        </xs:element>
        <xs:element name="database">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="provider"/>
                                <xs:element ref="dataSource"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:element name="properties">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="property" minOccurs="0" maxOccurs="unbounded"/>
                        </xs:sequence>
                        <xs:attribute name="resource" type="xs:string"/>
                        <xs:attribute name="url" type="xs:string"/>
                        <xs:attribute name="embedded" type="xs:string"/>
                </xs:complexType>
        </xs:element>
```

```
	<xs:element name="property">
		<xs:complexType>
			<xs:attribute name="resource" type="xs:string"/>
			<xs:attribute name="url" type="xs:string"/>
			<xs:attribute name="embedded" type="xs:string"/>
			<xs:attribute name="key" type="xs:string"/>
			<xs:attribute name="value" type="xs:string"/>
		</xs:complexType>
	</xs:element>
	<xs:element name="provider">
		<xs:complexType>
			<xs:attribute name="name" type="xs:string" use="required"/>
		</xs:complexType>
	</xs:element>
	<xs:element name="setting">
		<xs:complexType>
			<xs:attribute name="useStatementNamespaces" type="xs:string"  default="false"/>
			<xs:attribute name="cacheModelsEnabled" type="xs:string"/>
			<xs:attribute name="validateSqlMap" type="xs:string" default="false"/>
			<xs:attribute name="useEmbedStatementParams" type="xs:boolean" default="false"/>
			<xs:attribute name="useReflectionOptimizer" type="xs:boolean" default="true"/>
		</xs:complexType>
	</xs:element>
	<xs:element name="settings">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="setting" maxOccurs="unbounded"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
	<xs:element name="sqlMap">
		<xs:complexType>
			<xs:attribute name="resource" type="xs:string"/>
			<xs:attribute name="url" type="xs:string"/>
			<xs:attribute name="embedded" type="xs:string"/>
		</xs:complexType>
	</xs:element>
	<xs:element name="providers">
		<xs:complexType>
			<xs:attribute name="resource" type="xs:string"/>
			<xs:attribute name="url" type="xs:string"/>
			<xs:attribute name="embedded" type="xs:string"/>
		</xs:complexType>
	</xs:element>
	<xs:annotation>
		<xs:documentation>
		 The document root.
		</xs:documentation>
	</xs:annotation>
	<xs:element name="sqlMapConfig">
		<xs:complexType mixed="true">
			<xs:sequence>
				<xs:element ref="properties" minOccurs="0"/>
				<xs:element ref="settings" minOccurs="0"/>
				<xs:element ref="providers" minOccurs="0"/>
				<xs:element ref="database" minOccurs="0"/>
				<xs:element ref="alias" minOccurs="0"/>
				<xs:element ref="typeHandlers" minOccurs="0"/>
				<xs:element ref="sqlMaps" minOccurs="0"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
	<xs:element name="sqlMaps">
		<xs:complexType>
			<xs:sequence>
				<xs:element ref="sqlMap" maxOccurs="unbounded"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
</xs:schema>
```

Appendix B. iBATIS.NET's `SqlMap.xsd`

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
targetNamespace="http://ibatis.apache.org/mapping"
elementFormDefault="qualified"
xmlns:mstns="http://tempuri.org/XMLSchema.xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="http://ibatis.apache.org/mapping"
xmlns:vs="http://schemas.microsoft.com/Visual-Studio-Intellisense"
vs:friendlyname="iBATIS.NET mapping file Configuration Schema"
vs:ishtmlschema="false"
vs:iscasesensitive="true"
vs:requireattributequotes="true"
vs:defaultnamespacequalifier=""
vs:defaultnsprefix="" >

        <xs:annotation>
                <xs:documentation>
                Mapping XML Schema Definition
        </xs:documentation>
        </xs:annotation>

        <xs:element name="alias">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="typeAlias" maxOccurs="unbounded"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>

        <xs:element name="discriminator">
                <xs:complexType>
                        <xs:attribute name="column" type="xs:string"/>
                        <xs:attribute name="type" type="xs:string"/>
                        <xs:attribute name="typeHandler" type="xs:string"/>
                </xs:complexType>
        </xs:element>

        <xs:element name="subMap">
                <xs:complexType>
                        <xs:attribute name="value" use="required" type="xs:string"/>
                        <xs:attribute name="resultMapping" use="required" type="xs:string"/>
                </xs:complexType>
        </xs:element>

        <xs:element name="cacheModel">
                <xs:complexType>
                        <xs:sequence maxOccurs="unbounded">
                                <xs:element ref="flushInterval" minOccurs="0"/>
                                <xs:element ref="flushOnExecute" minOccurs="0" maxOccurs="unbounded"/>
                                <xs:element ref="property" minOccurs="0" maxOccurs="unbounded"/>
                        </xs:sequence>
                        <xs:attribute name="id" type="xs:string" use="required"/>
                        <xs:attribute name="serialize" type="xs:string"  default="false"/>
                        <xs:attribute name="readOnly" type="xs:string"  default="true"/>
                        <xs:attribute name="implementation" use="required">
                                <xs:simpleType>
                                        <xs:restriction base="xs:NMTOKEN">
                                                <xs:enumeration value="LRU"/>
                                                <xs:enumeration value="MEMORY"/>
                                                <xs:enumeration value="FIFO"/>
                                        </xs:restriction>
                                </xs:simpleType>
                        </xs:attribute>
                </xs:complexType>
        </xs:element>
        <xs:element name="cacheModels">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="cacheModel" maxOccurs="unbounded"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:element name="procedure">
                <xs:complexType mixed="true">
```

```
                    <xs:attribute name="id" type="xs:string" use="required"/>
                    <xs:attribute name="parameterMap" type="xs:string" use="required"/>
                    <xs:attribute name="resultMap" type="xs:string"/>
                    <xs:attribute name="resultClass" type="xs:string"/>
                    <xs:attribute name="cacheModel" type="xs:string"/>
                    <xs:attribute name="listClass" type="xs:string"/>
            </xs:complexType>
    </xs:element>
    <xs:element name="delete">
            <xs:complexType mixed="true">
                    <xs:choice minOccurs="0" maxOccurs="unbounded">
                            <xs:element ref="generate"/>
                            <xs:element ref="isEmpty"/>
                            <xs:element ref="isEqual"/>
                            <xs:element ref="isGreaterEqual"/>
                            <xs:element ref="isGreaterThan"/>
                            <xs:element ref="isLessThan"/>
                            <xs:element ref="isLessEqual"/>
                            <xs:element ref="isNotEmpty"/>
                            <xs:element ref="isNotEqual"/>
                            <xs:element ref="isNotNull"/>
                            <xs:element ref="isNotParameterPresent"/>
                            <xs:element ref="isNotPropertyAvailable"/>
                            <xs:element ref="isNull"/>
                            <xs:element ref="isParameterPresent"/>
                            <xs:element ref="isPropertyAvailable"/>
                            <xs:element ref="iterate"/>
                    </xs:choice>
                    <xs:attribute name="id" use="required"/>
                    <xs:attribute name="parameterMap" type="xs:string"/>
                    <xs:attribute name="parameterClass" type="xs:string"/>
                    <xs:attribute name="extends" type="xs:string"/>
                    <xs:attribute name="remapResults">
                            <xs:simpleType>
                                    <xs:restriction base="xs:NMTOKEN">
                                            <xs:enumeration value="false"/>
                                            <xs:enumeration value="true"/>
                                    </xs:restriction>
                            </xs:simpleType>
                    </xs:attribute>
            </xs:complexType>
    </xs:element>
    <xs:element name="dynamic">
            <xs:complexType mixed="true">
                    <xs:choice minOccurs="0" maxOccurs="unbounded">
                            <xs:element ref="isEmpty"/>
                            <xs:element ref="isEqual"/>
                            <xs:element ref="isGreaterEqual"/>
                            <xs:element ref="isGreaterThan"/>
                            <xs:element ref="isLessThan"/>
                            <xs:element ref="isLessEqual"/>
                            <xs:element ref="isNotEmpty"/>
                            <xs:element ref="isNotEqual"/>
                            <xs:element ref="isNotNull"/>
                            <xs:element ref="isNotParameterPresent"/>
                            <xs:element ref="isNotPropertyAvailable"/>
                            <xs:element ref="isNull"/>
                            <xs:element ref="isParameterPresent"/>
                            <xs:element ref="isPropertyAvailable"/>
                            <xs:element ref="iterate"/>
                    </xs:choice>
                    <xs:attribute name="prepend" type="xs:string"/>
            </xs:complexType>
    </xs:element>
    <xs:element name="flushInterval">
            <xs:complexType>
                    <xs:attribute name="milliseconds" type="xs:byte"/>
                    <xs:attribute name="seconds" type="xs:byte"/>
                    <xs:attribute name="minutes" type="xs:byte"/>
                    <xs:attribute name="hours" type="xs:byte"/>
            </xs:complexType>
    </xs:element>
    <xs:element name="flushOnExecute">
            <xs:complexType>
                    <xs:attribute name="statement" type="xs:string" use="required"/>
            </xs:complexType>
    </xs:element>
    <xs:element name="generate">
```

```
		<xs:complexType>
			<xs:attribute name="table" type="xs:string" use="required"/>
			<xs:attribute name="by" type="xs:string"/>
		</xs:complexType>
	</xs:element>
	<xs:element name="insert">
		<xs:complexType mixed="true">
			<xs:choice minOccurs="0" maxOccurs="unbounded">
				<xs:element ref="selectKey"/>
				<xs:element ref="generate"/>
				<xs:element ref="dynamic"/>
				<xs:element ref="isEmpty"/>
				<xs:element ref="isEqual"/>
				<xs:element ref="isGreaterEqual"/>
				<xs:element ref="isGreaterThan"/>
				<xs:element ref="isLessThan"/>
				<xs:element ref="isLessEqual"/>
				<xs:element ref="isNotEmpty"/>
				<xs:element ref="isNotEqual"/>
				<xs:element ref="isNotNull"/>
				<xs:element ref="isNotParameterPresent"/>
				<xs:element ref="isNotPropertyAvailable"/>
				<xs:element ref="isNull"/>
				<xs:element ref="isParameterPresent"/>
				<xs:element ref="isPropertyAvailable"/>
				<xs:element ref="iterate"/>
			</xs:choice>
			<xs:attribute name="id" type="xs:string" use="required"/>
			<xs:attribute name="parameterClass" type="xs:string"/>
			<xs:attribute name="parameterMap" type="xs:string"/>
			<xs:attribute name="resultClass" type="xs:string"/>
			<xs:attribute name="remapResults">
				<xs:simpleType>
					<xs:restriction base="xs:NMTOKEN">
						<xs:enumeration value="false"/>
						<xs:enumeration value="true"/>
					</xs:restriction>
				</xs:simpleType>
			</xs:attribute>
		</xs:complexType>
	</xs:element>
	<xs:element name="isNotParameterPresent">
		<xs:complexType mixed="true">
			<xs:choice minOccurs="0" maxOccurs="unbounded">
				<xs:element ref="isEmpty"/>
				<xs:element ref="isEqual"/>
				<xs:element ref="isGreaterEqual"/>
				<xs:element ref="isGreaterThan"/>
				<xs:element ref="isLessThan"/>
				<xs:element ref="isLessEqual"/>
				<xs:element ref="isNotEmpty"/>
				<xs:element ref="isNotEqual"/>
				<xs:element ref="isNotNull"/>
				<xs:element ref="isNotParameterPresent"/>
				<xs:element ref="isNotPropertyAvailable"/>
				<xs:element ref="isNull"/>
				<xs:element ref="isParameterPresent"/>
				<xs:element ref="isPropertyAvailable"/>
				<xs:element ref="iterate"/>
			</xs:choice>
			<xs:attribute name="prepend" type="xs:string"/>
		</xs:complexType>
	</xs:element>
	<xs:element name="isNotPropertyAvailable">
		<xs:complexType mixed="true">
			<xs:choice minOccurs="0" maxOccurs="unbounded">
				<xs:element ref="isEmpty"/>
				<xs:element ref="isEqual"/>
				<xs:element ref="isGreaterEqual"/>
				<xs:element ref="isGreaterThan"/>
				<xs:element ref="isLessThan"/>
				<xs:element ref="isLessEqual"/>
				<xs:element ref="isNotEmpty"/>
				<xs:element ref="isNotEqual"/>
				<xs:element ref="isNotNull"/>
				<xs:element ref="isNotParameterPresent"/>
				<xs:element ref="isNotPropertyAvailable"/>
				<xs:element ref="isNull"/>
```

```
                        <xs:element ref="isParameterPresent"/>
                        <xs:element ref="isPropertyAvailable"/>
                        <xs:element ref="iterate"/>
                </xs:choice>
                <xs:attribute name="prepend" type="xs:string"/>
                <xs:attribute name="property" type="xs:string" use="required"/>
        </xs:complexType>
</xs:element>
<xs:element name="isEmpty">
        <xs:complexType mixed="true">
                <xs:choice minOccurs="0" maxOccurs="unbounded">
                        <xs:element ref="isEmpty"/>
                        <xs:element ref="isEqual"/>
                        <xs:element ref="isGreaterEqual"/>
                        <xs:element ref="isGreaterThan"/>
                        <xs:element ref="isLessThan"/>
                        <xs:element ref="isLessEqual"/>
                        <xs:element ref="isNotEmpty"/>
                        <xs:element ref="isNotEqual"/>
                        <xs:element ref="isNotNull"/>
                        <xs:element ref="isNotParameterPresent"/>
                        <xs:element ref="isNotPropertyAvailable"/>
                        <xs:element ref="isNull"/>
                        <xs:element ref="isParameterPresent"/>
                        <xs:element ref="isPropertyAvailable"/>
                        <xs:element ref="iterate"/>
                </xs:choice>
                <xs:attribute name="prepend" type="xs:string"/>
                <xs:attribute name="property" type="xs:string"/>
        </xs:complexType>
</xs:element>
<xs:element name="isEqual">
        <xs:complexType mixed="true">
                <xs:choice minOccurs="0" maxOccurs="unbounded">
                        <xs:element ref="isEmpty"/>
                        <xs:element ref="isEqual"/>
                        <xs:element ref="isGreaterEqual"/>
                        <xs:element ref="isGreaterThan"/>
                        <xs:element ref="isLessThan"/>
                        <xs:element ref="isLessEqual"/>
                        <xs:element ref="isNotEmpty"/>
                        <xs:element ref="isNotEqual"/>
                        <xs:element ref="isNotNull"/>
                        <xs:element ref="isNotParameterPresent"/>
                        <xs:element ref="isNotPropertyAvailable"/>
                        <xs:element ref="isNull"/>
                        <xs:element ref="isParameterPresent"/>
                        <xs:element ref="isPropertyAvailable"/>
                        <xs:element ref="iterate"/>
                </xs:choice>
                <xs:attribute name="prepend" type="xs:string"/>
                <xs:attribute name="property" type="xs:string"/>
                <xs:attribute name="compareProperty" type="xs:string"/>
                <xs:attribute name="compareValue" type="xs:string"/>
        </xs:complexType>
</xs:element>
<xs:element name="isNull">
        <xs:complexType mixed="true">
                <xs:choice minOccurs="0" maxOccurs="unbounded">
                        <xs:element ref="isEmpty"/>
                        <xs:element ref="isEqual"/>
                        <xs:element ref="isGreaterEqual"/>
                        <xs:element ref="isGreaterThan"/>
                        <xs:element ref="isLessThan"/>
                        <xs:element ref="isLessEqual"/>
                        <xs:element ref="isNotEmpty"/>
                        <xs:element ref="isNotEqual"/>
                        <xs:element ref="isNotNull"/>
                        <xs:element ref="isNotParameterPresent"/>
                        <xs:element ref="isNotPropertyAvailable"/>
                        <xs:element ref="isNull"/>
                        <xs:element ref="isParameterPresent"/>
                        <xs:element ref="isPropertyAvailable"/>
                        <xs:element ref="iterate"/>
                </xs:choice>
                <xs:attribute name="prepend" type="xs:string"/>
                <xs:attribute name="property" type="xs:string"/>
        </xs:complexType>
```

```
	</xs:element>
	<xs:element name="isGreaterEqual">
		<xs:complexType mixed="true">
			<xs:choice minOccurs="0" maxOccurs="unbounded">
				<xs:element ref="isEmpty"/>
				<xs:element ref="isEqual"/>
				<xs:element ref="isGreaterEqual"/>
				<xs:element ref="isGreaterThan"/>
				<xs:element ref="isLessThan"/>
				<xs:element ref="isLessEqual"/>
				<xs:element ref="isNotEmpty"/>
				<xs:element ref="isNotEqual"/>
				<xs:element ref="isNotNull"/>
				<xs:element ref="isNotParameterPresent"/>
				<xs:element ref="isNotPropertyAvailable"/>
				<xs:element ref="isNull"/>
				<xs:element ref="isParameterPresent"/>
				<xs:element ref="isPropertyAvailable"/>
				<xs:element ref="iterate"/>
			</xs:choice>
			<xs:attribute name="prepend" type="xs:string"/>
			<xs:attribute name="property" type="xs:string"/>
			<xs:attribute name="compareProperty" type="xs:string"/>
			<xs:attribute name="compareValue" type="xs:string"/>
		</xs:complexType>
	</xs:element>
	<xs:element name="isGreaterThan">
		<xs:complexType mixed="true">
			<xs:choice minOccurs="0" maxOccurs="unbounded">
				<xs:element ref="isEmpty"/>
				<xs:element ref="isEqual"/>
				<xs:element ref="isGreaterEqual"/>
				<xs:element ref="isGreaterThan"/>
				<xs:element ref="isLessThan"/>
				<xs:element ref="isLessEqual"/>
				<xs:element ref="isNotEmpty"/>
				<xs:element ref="isNotEqual"/>
				<xs:element ref="isNotNull"/>
				<xs:element ref="isNotParameterPresent"/>
				<xs:element ref="isNotPropertyAvailable"/>
				<xs:element ref="isNull"/>
				<xs:element ref="isParameterPresent"/>
				<xs:element ref="isPropertyAvailable"/>
				<xs:element ref="iterate"/>
			</xs:choice>
			<xs:attribute name="prepend" type="xs:string"/>
			<xs:attribute name="property" type="xs:string"/>
			<xs:attribute name="compareProperty" type="xs:string"/>
			<xs:attribute name="compareValue" type="xs:string"/>
		</xs:complexType>
	</xs:element>
	<xs:element name="isLessEqual">
		<xs:complexType mixed="true">
			<xs:choice minOccurs="0" maxOccurs="unbounded">
				<xs:element ref="isEmpty"/>
				<xs:element ref="isEqual"/>
				<xs:element ref="isGreaterEqual"/>
				<xs:element ref="isGreaterThan"/>
				<xs:element ref="isLessThan"/>
				<xs:element ref="isLessEqual"/>
				<xs:element ref="isNotEmpty"/>
				<xs:element ref="isNotEqual"/>
				<xs:element ref="isNotNull"/>
				<xs:element ref="isNotParameterPresent"/>
				<xs:element ref="isNotPropertyAvailable"/>
				<xs:element ref="isNull"/>
				<xs:element ref="isParameterPresent"/>
				<xs:element ref="isPropertyAvailable"/>
				<xs:element ref="iterate"/>
			</xs:choice>
			<xs:attribute name="prepend" type="xs:string"/>
			<xs:attribute name="property" type="xs:string"/>
			<xs:attribute name="compareProperty" type="xs:string"/>
			<xs:attribute name="compareValue" type="xs:string"/>
		</xs:complexType>
	</xs:element>
	<xs:element name="isLessThan">
		<xs:complexType mixed="true">
```

```
                    <xs:choice minOccurs="0" maxOccurs="unbounded">
                            <xs:element ref="isEmpty"/>
                            <xs:element ref="isEqual"/>
                            <xs:element ref="isGreaterEqual"/>
                            <xs:element ref="isGreaterThan"/>
                            <xs:element ref="isLessThan"/>
                            <xs:element ref="isLessEqual"/>
                            <xs:element ref="isNotEmpty"/>
                            <xs:element ref="isNotEqual"/>
                            <xs:element ref="isNotNull"/>
                            <xs:element ref="isNotParameterPresent"/>
                            <xs:element ref="isNotPropertyAvailable"/>
                            <xs:element ref="isNull"/>
                            <xs:element ref="isParameterPresent"/>
                            <xs:element ref="isPropertyAvailable"/>
                            <xs:element ref="iterate"/>
                    </xs:choice>
                    <xs:attribute name="prepend" type="xs:string"/>
                    <xs:attribute name="property" type="xs:string"/>
                    <xs:attribute name="compareProperty" type="xs:string"/>
                    <xs:attribute name="compareValue" type="xs:string"/>
            </xs:complexType>
    </xs:element>
    <xs:element name="isNotEmpty">
            <xs:complexType mixed="true">
                    <xs:choice minOccurs="0" maxOccurs="unbounded">
                            <xs:element ref="isEmpty"/>
                            <xs:element ref="isEqual"/>
                            <xs:element ref="isGreaterEqual"/>
                            <xs:element ref="isGreaterThan"/>
                            <xs:element ref="isLessThan"/>
                            <xs:element ref="isLessEqual"/>
                            <xs:element ref="isNotEmpty"/>
                            <xs:element ref="isNotEqual"/>
                            <xs:element ref="isNotNull"/>
                            <xs:element ref="isNotParameterPresent"/>
                            <xs:element ref="isNotPropertyAvailable"/>
                            <xs:element ref="isNull"/>
                            <xs:element ref="isParameterPresent"/>
                            <xs:element ref="isPropertyAvailable"/>
                            <xs:element ref="iterate"/>
                    </xs:choice>
                    <xs:attribute name="prepend" type="xs:string"/>
                    <xs:attribute name="property" type="xs:string"/>
            </xs:complexType>
    </xs:element>
    <xs:element name="isNotEqual">
            <xs:complexType mixed="true">
                    <xs:choice minOccurs="0" maxOccurs="unbounded">
                            <xs:element ref="isEmpty"/>
                            <xs:element ref="isEqual"/>
                            <xs:element ref="isGreaterEqual"/>
                            <xs:element ref="isGreaterThan"/>
                            <xs:element ref="isLessThan"/>
                            <xs:element ref="isLessEqual"/>
                            <xs:element ref="isNotEmpty"/>
                            <xs:element ref="isNotEqual"/>
                            <xs:element ref="isNotNull"/>
                            <xs:element ref="isNotParameterPresent"/>
                            <xs:element ref="isNotPropertyAvailable"/>
                            <xs:element ref="isNull"/>
                            <xs:element ref="isParameterPresent"/>
                            <xs:element ref="isPropertyAvailable"/>
                            <xs:element ref="iterate"/>
                    </xs:choice>
                    <xs:attribute name="prepend" type="xs:string"/>
                    <xs:attribute name="property" type="xs:string"/>
                    <xs:attribute name="compareProperty" type="xs:string"/>
                    <xs:attribute name="compareValue" type="xs:string"/>
            </xs:complexType>
    </xs:element>
    <xs:element name="isNotNull">
            <xs:complexType mixed="true">
                    <xs:choice minOccurs="0" maxOccurs="unbounded">
                            <xs:element ref="isEmpty"/>
                            <xs:element ref="isEqual"/>
                            <xs:element ref="isGreaterEqual"/>
                            <xs:element ref="isGreaterThan"/>
```

```
                        <xs:element ref="isLessThan"/>
                        <xs:element ref="isLessEqual"/>
                        <xs:element ref="isNotEmpty"/>
                        <xs:element ref="isNotEqual"/>
                        <xs:element ref="isNotNull"/>
                        <xs:element ref="isNotParameterPresent"/>
                        <xs:element ref="isNotPropertyAvailable"/>
                        <xs:element ref="isNull"/>
                        <xs:element ref="isParameterPresent"/>
                        <xs:element ref="isPropertyAvailable"/>
                        <xs:element ref="iterate"/>
                </xs:choice>
                <xs:attribute name="prepend" type="xs:string"/>
                <xs:attribute name="property" type="xs:string"/>
        </xs:complexType>
</xs:element>
<xs:element name="isParameterPresent">
        <xs:complexType mixed="true">
                <xs:choice minOccurs="0" maxOccurs="unbounded">
                        <xs:element ref="isEmpty"/>
                        <xs:element ref="isEqual"/>
                        <xs:element ref="isGreaterEqual"/>
                        <xs:element ref="isGreaterThan"/>
                        <xs:element ref="isLessThan"/>
                        <xs:element ref="isLessEqual"/>
                        <xs:element ref="isNotEmpty"/>
                        <xs:element ref="isNotEqual"/>
                        <xs:element ref="isNotNull"/>
                        <xs:element ref="isNotParameterPresent"/>
                        <xs:element ref="isNotPropertyAvailable"/>
                        <xs:element ref="isNull"/>
                        <xs:element ref="isParameterPresent"/>
                        <xs:element ref="isPropertyAvailable"/>
                        <xs:element ref="iterate"/>
                </xs:choice>
                <xs:attribute name="prepend" type="xs:string"/>
        </xs:complexType>
</xs:element>
<xs:element name="isPropertyAvailable">
        <xs:complexType mixed="true">
                <xs:choice minOccurs="0" maxOccurs="unbounded">
                        <xs:element ref="isEmpty"/>
                        <xs:element ref="isEqual"/>
                        <xs:element ref="isGreaterEqual"/>
                        <xs:element ref="isGreaterThan"/>
                        <xs:element ref="isLessThan"/>
                        <xs:element ref="isLessEqual"/>
                        <xs:element ref="isNotEmpty"/>
                        <xs:element ref="isNotEqual"/>
                        <xs:element ref="isNotNull"/>
                        <xs:element ref="isNotParameterPresent"/>
                        <xs:element ref="isNotPropertyAvailable"/>
                        <xs:element ref="isNull"/>
                        <xs:element ref="isParameterPresent"/>
                        <xs:element ref="isPropertyAvailable"/>
                        <xs:element ref="iterate"/>
                </xs:choice>
                <xs:attribute name="prepend" type="xs:string"/>
                <xs:attribute name="property" type="xs:string" use="required"/>
        </xs:complexType>
</xs:element>
<xs:element name="iterate">
        <xs:complexType mixed="true">
                <xs:choice minOccurs="0" maxOccurs="unbounded">
                        <xs:element ref="isEmpty"/>
                        <xs:element ref="isEqual"/>
                        <xs:element ref="isGreaterEqual"/>
                        <xs:element ref="isGreaterThan"/>
                        <xs:element ref="isLessThan"/>
                        <xs:element ref="isLessEqual"/>
                        <xs:element ref="isNotEmpty"/>
                        <xs:element ref="isNotEqual"/>
                        <xs:element ref="isNotNull"/>
                        <xs:element ref="isNotParameterPresent"/>
                        <xs:element ref="isNotPropertyAvailable"/>
                        <xs:element ref="isNull"/>
                        <xs:element ref="isParameterPresent"/>
                        <xs:element ref="isPropertyAvailable"/>
```

```
                        <xs:element ref="iterate"/>
                </xs:choice>
                <xs:attribute name="open" type="xs:string" use="required"/>
                <xs:attribute name="close" type="xs:string" use="required"/>
                <xs:attribute name="conjunction" type="xs:string" use="required"/>
                <xs:attribute name="property" type="xs:string"/>
                <xs:attribute name="prepend" type="xs:string"/>
        </xs:complexType>
</xs:element>
<xs:element name="parameter">
        <xs:complexType>
                <xs:attribute name="property" type="xs:string" use="required"/>
                <xs:attribute name="column" type="xs:string"/>
                <xs:attribute name="nullValue" type="xs:string"/>
                <xs:attribute name="type" type="xs:string"/>
                <xs:attribute name="dbType" type="xs:string"/>
                <xs:attribute name="size" type="xs:string"/>
                <xs:attribute name="scale" type="xs:string"/>
                <xs:attribute name="precision" type="xs:string"/>
                <xs:attribute name="typeHandler" type="xs:string"/>
                <xs:attribute name="direction">
                        <xs:simpleType>
                                <xs:restriction base="xs:NMTOKEN">
                                        <xs:enumeration value="Input"/>
                                        <xs:enumeration value="Output"/>
                                        <xs:enumeration value="InputOutput"/>
                                </xs:restriction>
                        </xs:simpleType>
                </xs:attribute>
        </xs:complexType>
</xs:element>
<xs:element name="parameterMap">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="parameter" minOccurs="0" maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:attribute name="id" type="xs:string" use="required"/>
                <xs:attribute name="class" type="xs:string" />
                <xs:attribute name="extends" type="xs:string"/>
        </xs:complexType>
</xs:element>
<xs:element name="parameterMaps">
        <xs:complexType>
                <xs:sequence>
                        <xs:element ref="parameterMap" maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
</xs:element>
<xs:element name="property">
        <xs:complexType>
                <xs:attribute name="name" type="xs:string" use="required"/>
                <xs:attribute name="value" type="xs:string" use="required"/>
        </xs:complexType>
</xs:element>
<xs:element name="result">
        <xs:complexType>
                <xs:attribute name="property" type="xs:string" use="required"/>
                <xs:attribute name="column" type="xs:string"/>
                <xs:attribute name="lazyLoad">
                        <xs:simpleType>
                                <xs:restriction base="xs:NMTOKEN">
                                        <xs:enumeration value="false"/>
                                        <xs:enumeration value="true"/>
                                </xs:restriction>
                        </xs:simpleType>
                </xs:attribute>
                <xs:attribute name="select" type="xs:string"/>
                <xs:attribute name="nullValue" type="xs:string"/>
                <xs:attribute name="type" type="xs:string"/>
                <xs:attribute name="dbType" type="xs:string"/>
                <xs:attribute name="columnIndex" type="xs:string"/>
                <xs:attribute name="resultMapping" type="xs:string"/>
                <xs:attribute name="typeHandler" type="xs:string"/>
        </xs:complexType>
</xs:element>
<xs:element name="argument">
        <xs:complexType>
                <xs:attribute name="argumentName" type="xs:string" use="required"/>
```

```
                    <xs:attribute name="column" type="xs:string"/>
                    <xs:attribute name="select" type="xs:string"/>
                    <xs:attribute name="nullValue" type="xs:string"/>
                    <xs:attribute name="type" type="xs:string"/>
                    <xs:attribute name="dbType" type="xs:string"/>
                    <xs:attribute name="columnIndex" type="xs:string"/>
                    <xs:attribute name="resultMapping" type="xs:string"/>
                    <xs:attribute name="typeHandler" type="xs:string"/>
            </xs:complexType>
        </xs:element>
        <xs:element name="constructor">
            <xs:complexType>
                    <xs:sequence>
                            <xs:element ref="argument" maxOccurs="unbounded"/>
                    </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="resultMap">
            <xs:complexType>
                    <xs:sequence>
                            <xs:element ref="constructor" minOccurs="0" maxOccurs="1"/>
                            <xs:element ref="result" minOccurs="0" maxOccurs="unbounded"/>
                            <xs:element ref="discriminator" minOccurs="0" maxOccurs="1"/>
                            <xs:element ref="subMap" minOccurs="0" maxOccurs="unbounded"/>
                    </xs:sequence>
                    <xs:attribute name="id" type="xs:string" use="required"/>
                    <xs:attribute name="class" type="xs:string" use="required"/>
                    <xs:attribute name="extends" type="xs:string"/>
            </xs:complexType>
        </xs:element>
        <xs:element name="resultMaps">
            <xs:complexType>
                    <xs:sequence>
                            <xs:element ref="resultMap" maxOccurs="unbounded"/>
                    </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="select">
            <xs:complexType mixed="true">
                    <xs:choice minOccurs="0" maxOccurs="unbounded">
                            <xs:element ref="dynamic"/>
                            <xs:element ref="isEmpty"/>
                            <xs:element ref="isEqual"/>
                            <xs:element ref="isGreaterEqual"/>
                            <xs:element ref="isGreaterThan"/>
                            <xs:element ref="isLessThan"/>
                            <xs:element ref="isLessEqual"/>
                            <xs:element ref="isNotEmpty"/>
                            <xs:element ref="isNotEqual"/>
                            <xs:element ref="isNotNull"/>
                            <xs:element ref="isNotParameterPresent"/>
                            <xs:element ref="isNotPropertyAvailable"/>
                            <xs:element ref="isNull"/>
                            <xs:element ref="isParameterPresent"/>
                            <xs:element ref="isPropertyAvailable"/>
                            <xs:element ref="iterate"/>
                            <xs:element ref="generate"/>
                    </xs:choice>
                    <xs:attribute name="id" type="xs:string" use="required"/>
                    <xs:attribute name="parameterClass" type="xs:string"/>
                    <xs:attribute name="resultMap" type="xs:string"/>
                    <xs:attribute name="resultClass" type="xs:string"/>
                    <xs:attribute name="parameterMap" type="xs:string"/>
                    <xs:attribute name="cacheModel" type="xs:string"/>
                    <xs:attribute name="listClass" type="xs:string"/>
                    <xs:attribute name="extends" type="xs:string"/>
                    <xs:attribute name="remapResults">
                            <xs:simpleType>
                                    <xs:restriction base="xs:NMTOKEN">
                                            <xs:enumeration value="false"/>
                                            <xs:enumeration value="true"/>
                                    </xs:restriction>
                            </xs:simpleType>
                    </xs:attribute>
            </xs:complexType>
        </xs:element>
        <xs:element name="selectKey">
            <xs:complexType>
```

```
                    <xs:simpleContent>
                            <xs:extension base="xs:string">
                                    <xs:attribute name="property" type="xs:string" use="required"/>
                                    <xs:attribute name="type" use="required">
                                            <xs:simpleType>
                                                    <xs:restriction base="xs:NMTOKEN">
                                                            <xs:enumeration value="post"/>
                                                            <xs:enumeration value="pre"/>
                                                    </xs:restriction>
                                            </xs:simpleType>
                                    </xs:attribute>
                                    <xs:attribute name="resultClass" type="xs:string" use="required"/>
                            </xs:extension>
                    </xs:simpleContent>
            </xs:complexType>
    </xs:element>
    <xs:element name="sqlMap">
            <xs:complexType>
                    <xs:sequence>
                            <xs:element ref="alias" minOccurs="0"/>
                            <xs:element ref="cacheModels" minOccurs="0"/>
                            <xs:element ref="resultMaps" minOccurs="0"/>
                            <xs:element ref="statements" minOccurs="0" />
                            <xs:element ref="parameterMaps" minOccurs="0"/>
                    </xs:sequence>
                    <xs:attribute name="namespace" type="xs:string" use="required"/>
            </xs:complexType>
    </xs:element>
    <xs:element name="statement">
            <xs:complexType mixed="true">
                    <xs:choice minOccurs="0" maxOccurs="unbounded">
                            <xs:element ref="dynamic"/>
                            <xs:element ref="isEmpty"/>
                            <xs:element ref="isEqual"/>
                            <xs:element ref="isGreaterEqual"/>
                            <xs:element ref="isGreaterThan"/>
                            <xs:element ref="isLessThan"/>
                            <xs:element ref="isLessEqual"/>
                            <xs:element ref="isNotEmpty"/>
                            <xs:element ref="isNotEqual"/>
                            <xs:element ref="isNotNull"/>
                            <xs:element ref="isNotParameterPresent"/>
                            <xs:element ref="isNotPropertyAvailable"/>
                            <xs:element ref="isNull"/>
                            <xs:element ref="isParameterPresent"/>
                            <xs:element ref="isPropertyAvailable"/>
                            <xs:element ref="iterate"/>
                    </xs:choice>
                    <xs:attribute name="id" type="xs:string" use="required"/>
                    <xs:attribute name="parameterClass" type="xs:string"/>
                    <xs:attribute name="resultMap" type="xs:string"/>
                    <xs:attribute name="resultClass" type="xs:string"/>
                    <xs:attribute name="parameterMap" type="xs:string"/>
                    <xs:attribute name="listClass" type="xs:string"/>
                    <xs:attribute name="cacheModel" type="xs:string"/>
                    <xs:attribute name="remapResults">
                            <xs:simpleType>
                                    <xs:restriction base="xs:NMTOKEN">
                                            <xs:enumeration value="false"/>
                                            <xs:enumeration value="true"/>
                                    </xs:restriction>
                            </xs:simpleType>
                    </xs:attribute>
            </xs:complexType>
    </xs:element>
    <xs:element name="statements">
            <xs:complexType>
                    <xs:choice maxOccurs="unbounded">
                            <xs:element ref="statement"/>
                            <xs:element ref="insert"/>
                            <xs:element ref="update"/>
                            <xs:element ref="delete"/>
                            <xs:element ref="select"/>
                            <xs:element ref="procedure"/>
                    </xs:choice>
            </xs:complexType>
    </xs:element>
    <xs:element name="typeAlias">
```

```
                <xs:complexType>
                        <xs:attribute name="alias" type="xs:string" use="required"/>
                        <xs:attribute name="type" type="xs:string" use="required"/>
                </xs:complexType>
        </xs:element>
        <xs:element name="update">
                <xs:complexType mixed="true">
                        <xs:choice minOccurs="0" maxOccurs="unbounded">
                                <xs:element ref="generate"/>
                                <xs:element ref="dynamic"/>
                                <xs:element ref="isEmpty"/>
                                <xs:element ref="isEqual"/>
                                <xs:element ref="isGreaterEqual"/>
                                <xs:element ref="isGreaterThan"/>
                                <xs:element ref="isLessThan"/>
                                <xs:element ref="isLessEqual"/>
                                <xs:element ref="isNotEmpty"/>
                                <xs:element ref="isNotEqual"/>
                                <xs:element ref="isNotNull"/>
                                <xs:element ref="isNotParameterPresent"/>
                                <xs:element ref="isNotPropertyAvailable"/>
                                <xs:element ref="isNull"/>
                                <xs:element ref="isParameterPresent"/>
                                <xs:element ref="isPropertyAvailable"/>
                                <xs:element ref="iterate"/>
                        </xs:choice>
                        <xs:attribute name="id" type="xs:string" use="required"/>
                        <xs:attribute name="parameterMap" type="xs:string"/>
                        <xs:attribute name="parameterClass" type="xs:string"/>
                        <xs:attribute name="extends" type="xs:string"/>
                        <xs:attribute name="remapResults">
                                <xs:simpleType>
                                        <xs:restriction base="xs:NMTOKEN">
                                                <xs:enumeration value="false"/>
                                                <xs:enumeration value="true"/>
                                        </xs:restriction>
                                </xs:simpleType>
                        </xs:attribute>
                </xs:complexType>
        </xs:element>
</xs:schema>
```