LINQ to SQL – Mapping & Translation

Table of Contents
INTRODUCTION
LINQ TO SQL TYPES
SQL-CLR TYPE MAPPING
TYPE MAPPING RUN TIME BEHAVIOR MATRIX
ENUM MAPPING
DATETIME MAPPING
XML TYPE MAPPING
DECIMAL AND MONEY MAPPING
FLOATING POINT TYPES
BINARY AND STRING SERIALIZATION
BOOLEAN DATA TYPES (LINQ TO SQL)
UNSUPPORTED FUNCTIONALITY (LINQ TO SQL)
ATTRIBUTE-BASED MAPPING (LINQ TO SQL)
DATABASEATTRIBUTE
TABLEATTRIBUTE
COLUMNATTRIBUTE
ASSOCIATIONATTRIBUTE
INHERITANCEMAPPINGATTRIBUTE
FUNCTIONATTRIBUTE
PARAMETERATTRIBUTE
RESULTTYPEATTRIBUTE
DATAATTRIBUTE
CODE GENERATION IN LINQ TO SQL
DBML EXTRACTOR
CODE GENERATOR
XML SCHEMA DEFINITION FILE
EXTERNAL MAPPING REFERENCE (LINQ TO SQL)
STANDARD QUERY OPERATOR TRANSLATION (LINQ TO SQL)
CONCLUSION
Introduction

LINQ to SQL, a component of Visual Studio Code Name "Orcas", provides a run-time infrastructure for managing relational data as objects without losing the ability to query. It does this by translating language-integrated queries into SQL for execution by the database, and then translating the tabular results back into objects you define. Your application is then free to manipulate the objects while LINQ to SQL stays in the background tracking your changes automatically.

LINQ to SQL Types

SQOT (Standard Query Operator Translation) is a translation engine used in LINQ to SQL components that provides explicit conversion whereas Common Language Runtime (CLR) constructs have corresponding expressions in SQL only. Some functionality in the .NET Framework classes is not supported for translation to SQL. This limitation applies also to user-defined methods, properties, and casts.

However, you can use such unsupported functionality in queries if either of the following is true:

- The method can be evaluated to a translatable value before the conversion process occurs. In other words, the method must not depend on any lambda variables that are not bound until query execution occurs.
- The method can be applied to the results after it is retrieved from the database. In other words, no additional requests for database information can be made after the method is applied.

SQL-CLR Type Mapping

A type mapping is a pairing between the common-language runtime (CLR) type of a field or property of an object and the SQL Server type of a table field.

This topic provides a complete mapping matrix and some specific information about the following:
1. Enum, DateTime, and XML mapping.
2. SQL Server money types and CLR conversions.
3. Floating point types.
4. Binary and string serialization.

### Type Mapping Run-time Behavior Matrix

<table>
<thead>
<tr>
<th>Type</th>
<th>SQL → CLR</th>
<th>Boolean</th>
<th>Byte</th>
<th>Int16</th>
<th>Int32</th>
<th>Int64</th>
<th>SByte</th>
<th>UByte</th>
<th>UInt16</th>
<th>UInt32</th>
<th>UInt64</th>
<th>Decimal</th>
<th>Single</th>
<th>Double</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Integers</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>TINYINT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>INT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BIGINT</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Decimals</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SMALLMONEY</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MONEY</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DECIMAL(20)/NUMERIC(20)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DECIMAL(28)/NUMERIC(28)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DECIMAL(29)/NUMERIC(29)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DECIMAL(38)/NUMERIC(38)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Floating-point</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>REAL/FLOAT(24)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FLOAT/FLOAT(32)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FLOAT/FLOAT(64)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Red-outlined cells show the default SQL type mappings for each column’s CLR type.

**Timestamp:** SQL Server supports reading from the field but not writing to it.

Note: Above diagram show a detailed matrix of mappings between CLR types on the top and SQL Server types on the left. Each cell represents the run-time behavior that is expected when data of a SQL type is retrieved from or saved to fields or properties that have corresponding CLR types. LINQ to SQL does not support mapping to any CLR or SQL data types that are not specified in this matrix (except as enabled by string or binary serialization).
**Enum Mapping**
LINQ to SQL supports the mapping of CLR Enum types in two ways:

- **Mapping to SQL numeric types (TINYINT, SMALLINT, INT, BIGINT)**
  
  When you map a CLR Enum type to a SQL numeric type, the underlying integer value of the CLR Enum is mapped to the value of the SQL database field. The value of the field in SQL is retrieved directly as the underlying integral value of the Enum type. When you change the Enum value and data is saved to the database, the underlying integral value of the Enum is stored to the database field.

- **Mapping to SQL text types (CHAR, NCHAR, VARCHAR, NVARCHAR)**
  
  When you map a CLR Enum type to a SQL text type, the SQL database value is mapped to the names of the CLR Enum members. For example, if an Enum named DaysOfWeek contains a member named Tue, that member maps to a database value of Tue. You can accomplish this mapping by using reflection over the Enum type.

The default SQL mapping for a CLR Enum type is the SQL equivalent of its underlying integral type.

**DateTime Mapping**

DateTime values are saved as is to the database without TimeZone conversion, regardless of the original DateTimeKind information. When DateTime values are retrieved from the database, their value is loaded as is into a DateTime with a DateTimeKind of Unspecified. For more information, see System.DateTime Methods (LINQ to SQL).

**XML Type Mapping**

You can map the SQL Server 2005 XML data type to XElement (default) or String. If the column stores XML fragments that cannot be read into XElement, the column must be mapped to String to avoid run-time errors. Examples of XML fragments that must be mapped to String include the following:

- A sequence of XML elements.
- Attributes, PIs, comments.

**Note:** XDocument cannot be used to map XML data types because this class is not serializable (does not implement the IXmlSerializable interface).

**Decimal and Money Types**

The SQL Server DECIMAL/MONEY/SMALLMONEY types and the CLR Decimal/Double types have the following differences:

<table>
<thead>
<tr>
<th>Types</th>
<th>SQL Server</th>
<th>CLS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal (precision, scale)</td>
<td>Up to 38 digits of precision&lt;br&gt;Range (with all digits to left of decimal point) (-10^{38} + 1 \text{ to } 10^{38} - 1).&lt;br&gt;Can represent all possible 0-38 digit numbers.</td>
<td>Up to 28-29 digits of precision&lt;br&gt;Range (with all digits to left of decimal point) (-2^{96} + 1 \text{ to } 2^{96} - 1).&lt;br&gt;Can represent all possible 0-28 digit numbers, and some but not all 29-digit numbers.</td>
</tr>
<tr>
<td>Money</td>
<td>Up to 18-19 digits of precision, but always with exactly 4 digits right of decimal point.&lt;br&gt;Range: (-2^{63}/1000 \text{ to } (2^{63} - 1)/1000)&lt;br&gt;Can represent all possible 0-18 digit numbers, and some but not all 19-digit numbers</td>
<td>- N/A-</td>
</tr>
<tr>
<td>SMALLMONEY</td>
<td>Up to 5-6 digits of precision, but always with exactly 4 digits right of</td>
<td>- N/A-</td>
</tr>
</tbody>
</table>
### Floating Point Types
- SQL Server supports floating point type of variable size, specified as `FLOAT (mantissaBits)`.
- CLR `Single` is equivalent to `REAL`, a synonym for `FLOAT (24)`.
- CLR `Double` is equivalent to `FLOAT`, which defaults to `FLOAT (53)`.
- LINQ to SQL maps `FLOATs` that are `FLOAT (24)` or less to Single, and larger floats to `Double`.

### Binary and String Serialization
LINQ to SQL supports two types of serialization of .NET Framework classes and user classes.

1. String serialization (`Parse()`)

---

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Decimal point.</td>
<td>Range: (-2^{32}/1000) to ((2^{33} - 1)/1000).</td>
<td>Can represent all possible 0-5 digit numbers, and some but not all 6 digit numbers.</td>
</tr>
<tr>
<td>Double</td>
<td>-N/A-</td>
<td>Range (\pm4.94065645841246544\times10^{-324}) to (1.79769313486231570\times10^{308}). Supports much greater magnitude than Decimal but has less precision. All decimal values can be converted to Double without overflow, but precision can be lost.</td>
</tr>
</tbody>
</table>
If a class implements .Parse () (with a signature similar to DateTime, for example), you can serialize it to any SQL text field (CHAR, NCHAR, VARCHAR, NVARCHAR, TEXT, NTEXT, XML). If you serialize the object to a string, the value returned from ToString () is saved to the database.

If you deserialize a stored string, .Parse () is invoked on the string to return the constructed object.

2. Binary serialization (ISerializable)

If a class implements ISerializable, you can serialize it to any SQL binary field (BINARY, VARBINARY, IMAGE). The standard behavior of ISerializable is followed to serialize and deserialize the object.

3. IXmlSerializable

LINQ to SQL does not support serialization by using IXmlSerializable.

**Boolean Data Types (LINQ to SQL)**

Boolean operators work as expected in the common language runtime (CLR), except that short-circuiting behavior is not translated. For example, the Visual Basic AndAlso operator behaves like the And operator. The C# && operator behaves like the & operator.

**Unsupported Functionality (LINQ to SQL)**

The following SQL functionality cannot be exposed through translation of existing common language runtime (CLR) and .NET Framework constructs:

- LIKE
- STDDEV

Apart from the above, LINQ to SQL does not support some of the methods in .NET that are listed below for reference.

1. System.Convert methods
2. System.DateTime methods
3. System.Math methods
   - Math..::.DivRem(Int32, Int32, Int32%)
- Math.DivRem(Int64, Int64, Int64%)
- Math.IEEERemainder(Double, Double)

4. System.Object methods
5. System.String methods
6. System.TimeSpan methods

**Attribute-Based Mapping (LINQ to SQL)**

LINQ to SQL maps a SQL Server database to a LINQ to SQL object model by either applying attributes or by using an external mapping file. In Attribute-Based Mapping, most elementary form, LINQ to SQL maps a database to a DataContext, a table to a class, and columns and relationships to properties on those classes. You can also use attributes to map an inheritance hierarchy in your object model.

**Note:** Developers using Visual Studio typically perform attribute-based mapping by using the Object Relational Designer

**DatabaseAttribute Attribute**

Use this attribute to specify the default name of the database when a name is not supplied by the connection. This attribute is optional, but if you use it, you must apply the Name property, as described in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>Specifies the name of the database.</td>
</tr>
</tbody>
</table>

**TableAttribute Attribute**

Use this attribute to designate a class as an entity class that is associated with a database table or view. LINQ to SQL treats classes that have this attribute as persistent classes. The following table describes the Name property.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>Designates a class as an entity class associated with a database table.</td>
</tr>
</tbody>
</table>

**ColumnAttribute Attribute**

Use this attribute to designate a member of an entity class to represent a column in a database table. You can apply this attribute to any field or property. Only those
members you identify as columns are retrieved and persisted when LINQ to SQL saves changes to the database. Members without this attribute are assumed to be non-persistent and are not submitted for inserts or updates.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AutoSync</td>
<td>AutoSync</td>
<td>Instructs the common language runtime (CLR) to retrieve the value after an insert or update operation. Options: Always, Never, OnUpdate, OnInsert.</td>
</tr>
<tr>
<td>CanBeNull</td>
<td>Boolean</td>
<td>Indicates that a column can contain null values.</td>
</tr>
<tr>
<td>DbType</td>
<td>String</td>
<td>Uses database types and modifiers to specify the type of the database column.</td>
</tr>
<tr>
<td>Expression</td>
<td>String</td>
<td>Defines a computed column in a database.</td>
</tr>
<tr>
<td>IsDbGenerated</td>
<td>Boolean</td>
<td>Indicates that a column contains values that the database auto-generates.</td>
</tr>
<tr>
<td>IsDiscriminator</td>
<td>Boolean</td>
<td>Indicates that the column contains a discriminator value for a LINQ to SQL inheritance hierarchy.</td>
</tr>
<tr>
<td>IsPrimaryKey</td>
<td>Boolean</td>
<td>Specifies that this class member represents a column that is or is part of the primary keys of the table.</td>
</tr>
<tr>
<td>IsVersion</td>
<td>Boolean</td>
<td>Identifies the column type of the member as a database timestamp or version number.</td>
</tr>
<tr>
<td>UpdateCheck</td>
<td>UpdateCheck</td>
<td>Specifies how LINQ to SQL approaches the detection of optimistic concurrency conflicts.</td>
</tr>
</tbody>
</table>

**AssociationAttribute Attribute**

Use this attribute to designate a property to represent an association in the database, such as a foreign key to primary key relationship.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeleteOnNull</td>
<td>Boolean</td>
<td>When placed on an association whose foreign key members are all non-nullable, deletes the object when the association is set to null.</td>
</tr>
<tr>
<td>DeleteRule</td>
<td>String</td>
<td>Delete behavior to an association.</td>
</tr>
<tr>
<td>IsForeignKey</td>
<td>Boolean</td>
<td>If true, designates the member as the foreign key in an association representing a database relationship.</td>
</tr>
<tr>
<td>IsUnique</td>
<td>Boolean</td>
<td>If true, indicates a uniqueness constraint on the foreign key.</td>
</tr>
<tr>
<td>Property</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>OtherKey</td>
<td>String</td>
<td>Designates one or more members of the target entity class as key values on the other side of the association.</td>
</tr>
<tr>
<td>ThisKey</td>
<td>String</td>
<td>Designates members of this entity class to represent the key values on this side of the association.</td>
</tr>
</tbody>
</table>

### InheritanceMappingAttribute Attribute

Use this attribute to map an inheritance hierarchy.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>String</td>
<td>Specifies the code value of the discriminator.</td>
</tr>
<tr>
<td>IsDefault</td>
<td>Boolean</td>
<td>If true, instantiates an object of this type when no discriminator value in the store matches any one of the specified values.</td>
</tr>
<tr>
<td>Type</td>
<td>Type</td>
<td>Specifies the type of the class in the hierarchy.</td>
</tr>
</tbody>
</table>

### FunctionAttribute Attribute

Use this attribute to designate a method as representing a stored procedure or user-defined function in the database.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsComposable</td>
<td>Boolean</td>
<td>If false, indicates mapping to a stored procedure. If true, indicates mapping to a user-defined function.</td>
</tr>
<tr>
<td>Name</td>
<td>String</td>
<td>Specifies the name of the stored procedure or user-defined function.</td>
</tr>
</tbody>
</table>

### ParameterAttribute Attribute

Use this attribute to map input parameters on stored procedure methods.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DbType</td>
<td>String</td>
<td>Specifies database type.</td>
</tr>
<tr>
<td>Name</td>
<td>String</td>
<td>Specifies a name for the parameter.</td>
</tr>
</tbody>
</table>

### ResultTypeAttribute Attribute

Use this attribute to specify a result type.
<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Type</td>
<td>Used on methods mapped to stored procedures that return <code>IMultipleResults</code>. Declares the valid or expected type mappings for the stored procedure.</td>
</tr>
</tbody>
</table>

**DataAttribute Attribute**

Use this attribute to specify names and private storage fields.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
<td>Specifies the name of the table, column, and so on.</td>
</tr>
<tr>
<td>Storage</td>
<td>String</td>
<td>Specifies the name of the underlying storage field.</td>
</tr>
</tbody>
</table>

**Code Generation in LINQ to SQL**

You can generate code to represent a database by using either the Object Relational Designer or the SQLMetal command-line tool (SqlMetal.exe). In either case, end-to-end code generation occurs in three stages:

1. The DBML Extractor extracts schema information from the database and reassembles the information into an XML-formatted DBML file.
2. The DBML file is scanned by the DBML Validator for errors.
3. If no validation errors appear, the file is passed to the Code Generator.

**DBML Extractor**

The DBML Extractor is a LINQ to SQL component that takes database metadata as input and produces a DBML file as output. The following illustration points out the sequence of operations.

![DBML Extractor Diagram](image.png)
**Code Generator**
The Code Generator is a LINQ to SQL component that translates DBML files to Visual Basic, C#, or XML mapping files. The following illustration points out the sequence of operations.

![Code Generator Diagram]

**XML Schema Definition File**
The DBML file must be valid against the following schema definition as an XSD file.

Distinguish this schema definition file from the schema definition file that is used to validate an external mapping file.

**External Mapping Reference (LINQ to SQL)**
LINQ to SQL supports external mapping, a process by which you use a separate XML file to specify mapping between the data model of the database and your object model. Advantages of using an external mapping file include the following:

- You can keep your mapping code out of your application code. This approach reduces clutter in your application code.

- You can treat an external mapping file something like a configuration file. For example, you can update how your application behaves after shipping the binaries by just swapping out the external mapping file.

**Requirements**
The mapping file must be an XML file, and the file must validate against a LINQ to SQL schema definition (.xsd) file.

The following rules apply:
The mapping file must be an XML file.
The XML mapping file must be valid against the XML schema definition file.
For more information
External mapping overrides attribute-based mapping. In other words, when you use an external mapping source to create a DataContext, the DataContext ignores all mapping attributes you have created on classes. This behavior is true whether the class is included in the external mapping file.
LINQ to SQL does not support the hybrid use of the two mapping approaches (attribute-based and external).
External mapping can be specific to a database provider. You can map the same class by using separate external mappings for separate providers. This is a feature that attribute-based mapping does not support.

**Standard Query Operator Translation (LINQ to SQL)**

LINQ to SQL translates Standard Query Operators to SQL commands. The query processor of the database determines the execution semantics of SQL translation.

Standard Query Operators are defined against sequences. A sequence is ordered and relies on reference identity for each element of the sequence.

SQL deals primarily with unordered sets of values. Ordering is typically an explicitly stated, post-processing operation that is applied to the final result of a query rather than to intermediate results. Identity is defined by values. For this reason, SQL queries are understood to deal with multisets (bags) instead of sets.

The following paragraphs describe the differences between the Standard Query Operators and their SQL translation for the SQL Server provider for LINQ to SQL.

**Operator Support**

**Concat**

The `Concat< (Of TSource)>` method is defined for ordered multisets where the order of the receiver and the order of the argument are the same. `Concat< (Of TSource)>` works as **UNION ALL** over the multisets followed by the common order.
The final step is ordering in SQL before results are produced. `Concat<\(\langle{TSource}\rangle\)\>)` does not preserve the order of its arguments. To ensure appropriate ordering, you must explicitly order the results of `Concat<\(\langle{TSource}\rangle\)\>)`.

**Intersect, Except, Union**

The `Intersect` and `Except` methods are well defined only on sets. The semantics for multisets is undefined.

The `Union` method is defined for multisets as the unordered concatenation of the multisets (effectively the result of the UNION ALL clause in SQL).

**Take, Skip**

`Take<\(\langle{TSource}\rangle\)\>)` and `Skip<\(\langle{TSource}\rangle\)\>)` methods are well defined only against ordered sets. The semantics for unordered sets or multisets are undefined.

```csharp
var custQuery =
(from cust in db.Customers
where cust.City == "London"
orderby cust.CustomerID
select cust).Skip (1).Take(1);
```

The generated SQL for this code moves the ordering to the end, as follows:

```
SELECT TOP 1 [t0].[CustomerID], [t0].[CompanyName],
FROM [Customers] AS [t0]
WHERE (NOT (EXISTS(
    SELECT NULL AS [EMPTY]
    FROM (SELECT TOP 1 [t1].[CustomerID]
        FROM [Customers] AS [t1]
        WHERE [t1].[City] = @p0
        ORDER BY [t1].[CustomerID]
    ) AS [t2]
    WHERE [t0].[CustomerID] = [t2].[CustomerID]
  )) AND ([t0].[City] = @p1)
ORDER BY [t0].[CustomerID]
```

It becomes obvious that all the specified ordering must be consistent when `Take<\(\langle{TSource}\rangle\)\>)` and `Skip<\(\langle{TSource}\rangle\)\>)` are chained together. Otherwise, the results are undefined.
Both `Take<(Of <(TSource)>)>` and `Skip<(Of <(TSource)>)>` are well-defined for non-negative, constant integral arguments based on the Standard Query Operator specification.

**Operators with No Translation**

The following methods are not translated by LINQ to SQL. The most common reason is the difference between unordered multisets and sequences.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>TakeWhile, SkipWhile</code></td>
<td>SQL queries operate on multisets, not on sequences. ORDER BY must be the last clause applied to the results. For this reason, there is no general-purpose translation for these two methods.</td>
</tr>
<tr>
<td><code>Reverse&lt;(Of &lt;(TSource)&gt;)&gt;</code></td>
<td>Translation of this method is possible for an ordered set but is not currently translated by LINQ to SQL.</td>
</tr>
<tr>
<td><code>Last, LastOrDefault</code></td>
<td>Translation of these methods is possible for an ordered set but is not currently translated by LINQ to SQL.</td>
</tr>
<tr>
<td><code>ElementAt&lt;(Of &lt;(TSource)&gt;)&gt;, ElementAtOrDefault&lt;(Of &lt;(TSource)&gt;)&gt;</code></td>
<td>SQL queries operate on multisets, not on indexable sequences.</td>
</tr>
<tr>
<td><code>DefaultIfEmpty (overload with default arg)</code></td>
<td>In general, a default value cannot be specified for an arbitrary tuple. Null values for tuples are possible in some cases through outer joins.</td>
</tr>
</tbody>
</table>

**Expression Translation**

**Null semantics**

LINQ to SQL does not impose null comparison semantics on SQL. Comparison operators are syntactically translated to their SQL equivalents. For this reason, the semantics reflect SQL semantics that are defined by server or connection settings. For example, two null values are considered unequal under default SQL Server settings, but you can change the settings to change the semantics. LINQ to SQL does not consider server settings when it translates queries.

A comparison with the literal null is translated to the appropriate SQL version (`is null` or `is not null`).
The value of null in collation is defined by SQL Server. LINQ to SQL does not change the collation.

**Aggregates**
The Standard Query Operator aggregate method **Sum** evaluates to zero for an empty sequence or for a sequence that contains only nulls. In LINQ to SQL, the semantics of SQL are left unchanged, and **Sum** evaluates to null instead of zero for an empty sequence or for a sequence that contains only nulls.

SQL limitations on intermediate results apply to aggregates in LINQ to SQL. The **Sum** of 32-bit integer quantities is not computed by using 64-bit results. Overflow might occur for a LINQ to SQL translation of **Sum**, even if the Standard Query Operator implementation does not cause an overflow for the corresponding in-memory sequence.

Likewise, the LINQ to SQL translation of **Average** of integer values is computed as an **integer**, not as a **double**.

**Entity Arguments**
LINQ to SQL enables entity types to be used in the **GroupBy** and **OrderBy** methods. In the translation of these operators, the use of an argument of a type is considered to be the equivalent to specifying all members of that type. For example, the following code is equivalent:

```csharp
db.Customers.GroupBy(c => c);

db.Customers.GroupBy(c => new { c.CustomerID, c.ContactName });
```

**Equatable / Comparable Arguments**
Equality of arguments is required in the implementation of the following methods:

- Contains
- Skip<(Of <(TSource)>)>
- Union
- Intersect
- Except
LINQ to SQL supports equality and comparison for flat arguments, but not for arguments that are or contain sequences. A flat argument is a type that can be mapped to a SQL row. A projection of one or more entity types that can be statically determined not to contain a sequence is considered a flat argument.

Following are examples of flat arguments:

```csharp
db.Customers.Select(c => c);
db.Customers.Select(c => new { c.CustomerID, c.City });
db.Orders.Select(o => new { o.OrderID, o.Customer.City });
db.Orders.Select(o => new { o.OrderID, o.Customer });
```

The following are examples of non-flat (hierarchical) arguments.

```csharp
// In the following line, c.Orders is a sequence.
db.Customers.Select(c => new { c.CustomerID, c.Orders });

// In the following line, the result has a sequence.
db.Customers.GroupBy(c => c.City);
```

**Visual Basic Function Translation**

The following helper functions that are used by the Visual Basic compiler are translated to corresponding SQL operators and functions:

- `CompareString`
- `DateTime.Compare`
- `Decimal.Compare`
- `IIf` (in `Microsoft.VisualBasic.Interaction`)

**Conversion methods:**

<table>
<thead>
<tr>
<th>Conversion Table</th>
<th>ToSByte</th>
<th>ToByte</th>
<th>ToChar</th>
</tr>
</thead>
<tbody>
<tr>
<td>ToBoolean</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ToCharArrayRankOne</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ToInteger</td>
<td>ToUInteger</td>
<td>ToLong</td>
<td>ToULong</td>
</tr>
<tr>
<td>ToDate</td>
<td>ToDecimal</td>
<td>ToDouble</td>
<td></td>
</tr>
<tr>
<td>ToDecimal</td>
<td>ToDouble</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ToCharArrayRankOne</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ToBoolean</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Conversion Table**

<table>
<thead>
<tr>
<th>Conversion Table</th>
<th>ToSByte</th>
<th>ToByte</th>
<th>ToChar</th>
</tr>
</thead>
<tbody>
<tr>
<td>ToBoolean</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ToCharArrayRankOne</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ToInteger</td>
<td>ToUInteger</td>
<td>ToLong</td>
<td>ToULong</td>
</tr>
<tr>
<td>ToDate</td>
<td>ToDecimal</td>
<td>ToDouble</td>
<td></td>
</tr>
<tr>
<td>ToDecimal</td>
<td>ToDouble</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ToCharArrayRankOne</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Inheritance in Queries

C# casts are supported only in projection. Casts that are used elsewhere are not translated and are ignored. Aside from SQL function names, SQL really only performs the equivalent of the common language runtime (CLR) Convert. That is, SQL can change the value of one type to another. There is no equivalent of CLR cast because there is no concept of reinterpreting the same bits as those of another type. That is why a C# cast works only locally. It is not remoted.

The operators, is and as, and the GetType method are not restricted to the Select operator. They can be used in other query operators also.

### SQL Server 2005 Support

LINQ to SQL does not support the following SQL Server 2005 features:

- Stored procedures written for SQL CLR.
- User-defined type.
- XML query features.

### Cross Apply and Outer Apply Operators

These operators are not available in SQL Server 2000. LINQ to SQL tries a series of rewrites to replace them with appropriate joins.

**Cross Apply** and **Outer Apply** are generated for relationship navigations. The set of queries for which such rewrites are possible is not well defined. For this reason, the minimal set of queries that is supported for SQL Server 2000 is the set that does not involve relationship navigation.

### text / ntext

Data types **text** / **ntext** cannot be used in certain query operations against **varchar(max)** / **nvarchar(max)**, which are supported by Microsoft SQL Server 2005.

No resolution is available for this limitation. Specifically, you cannot use **Distinct()** on any result that contains members that are mapped to **text** or **ntext** columns.
Behavior Triggered by Nested Queries

SQL Server 2000 (through SP4) binder has some idiosyncrasies that are triggered by nested queries. The set of SQL queries that triggers these idiosyncrasies is not well defined. For this reason, you cannot define the set of LINQ to SQL queries that might cause SQL Server exceptions.

Skip and Take Operators

Take<(Of <(TSource)>)> and Skip<(Of <(TSource)>)> have certain limitations when they are used in queries against SQL Server 2000.

Object Materialization

Materialization creates CLR objects from rows that are returned by one or more SQL queries.

- The following calls are executed locally as a part of materialization:
  - Constructors
  - ToString methods in projections
  - Type casts in projections

- Methods that follow the AsEnumerable<(Of <(TSource)>)> method are executed locally. This method does not cause immediate execution.

- You can use a struct as the return type of a query result or as a member of the result type. Entities are required to be classes. Anonymous types are materialized as class instances, but named structs (non-entities) can be used in projection.

- A member of the return type of a query result can be of type IQueryable<(Of <(T)>)>. It is materialized as a local collection.

- The following methods cause the immediate materialization of the sequence that the methods are applied to:
  - ToList<(Of <(TSource)>)>
  - ToDictionary
  - ToArray<(Of <(TSource)>)>
Conclusion

Language-Integrated Query (LINQ) is a groundbreaking innovation in Visual Studio 2008 and the .NET Framework version 3.5 that bridges the gap between the world of objects and the world of data. By using LINQ to SQL, you can use the LINQ technology to access SQL databases just as you would access an in-memory collection.