Contents

DEREF

EXCEPT

EXISTS

FLATTEN

FROM

FUNCTION

GROUP BY

GROUPPARTITION

HAVING

KEY

IN

INTERSECT

ISNULL

ISOF

LIKE

LIMIT

MULTISET

Named Type Constructor

NAVIGATE

OFTYPE

ORDER BY

OVERLAPS

REF

ROW

SELECT

SET

SKIP

THEN

TOP

TREAT

UNION

USING

Entity SQL language reference

11/8/2022 • 2 minutes to read • Edit Online

This section provides detailed documentation LINQ to Entities, Entity SQL, and the modeling and mapping languages used by the Entity Framework.

In this section

- LINQ to Entities
- Entity SQL Language
- Canonical Functions

Related sections

- ADO.NET Entity Data Model Tools

See also

- ADO.NET Entity Framework
- Getting Started
- Samples
- CSDL, SSDL, and MSL Specifications

LINQ to Entities

11/8/2022 • 5 minutes to read • Edit Online

LINQ to Entities provides Language-Integrated Query (LINQ) support that enables developers to write queries against the Entity Framework conceptual model using Visual Basic or Visual C#. Queries against the Entity Framework are represented by command tree queries, which execute against the object context. LINQ to Entities converts Language-Integrated Queries (LINQ) queries to command tree queries, executes the queries against the Entity Framework, and returns objects that can be used by both the Entity Framework and LINQ. The following is the process for creating and executing a LINQ to Entities query:

1. Construct an ObjectQuery<T> instance from ObjectContext.
2. Compose a LINQ to Entities query in C# or Visual Basic by using the ObjectQuery<T> instance.
3. Convert LINQ standard query operators and expressions to command trees.
4. Execute the query, in command tree representation, against the data source. Any exceptions thrown on the data source during execution are passed directly up to the client.
5. Return query results back to the client.

Constructing an ObjectQuery Instance

The ObjectQuery<T> generic class represents a query that returns a collection of zero or more typed entities. An object query is typically constructed from an existing object context, instead of being manually constructed, and always belongs to that object context. This context provides the connection and metadata information that is required to compose and execute the query. The ObjectQuery<T> generic class implements the IQueryable<T> generic interface, whose builder methods enable LINQ queries to be incrementally built. You can also let the compiler infer the type of entities by using the C# `var` keyword (`Dim` in Visual Basic, with local type inference enabled).

Composing the Queries

Instances of the ObjectQuery<T> generic class, which implements the generic IQueryable<T> interface, serve as the data source for LINQ to Entities queries. In a query, you specify exactly the information that you want to retrieve from the data source. A query can also specify how that information should be sorted, grouped, and shaped before it is returned. In LINQ, a query is stored in a variable. This query variable takes no action and returns no data; it only stores the query information. After you create a query you must execute that query to retrieve any data.

LINQ to Entities queries can be composed in two different syntaxes: query expression syntax and method-based query syntax. Query expression syntax and method-based query syntax are new in C# 3.0 and Visual Basic 9.0.

For more information, see Queries in LINQ to Entities.

Query Conversion

To execute a LINQ to Entities query against the Entity Framework, the LINQ query must be converted to a command tree representation that can be executed against the Entity Framework.

LINQ to Entities queries are comprised of LINQ standard query operators (such as Select, Where, and GroupBy) and expressions (x > 10, Contact.LastName, and so on). LINQ operators are not defined by a class, but rather are

methods on a class. In LINQ, expressions can contain anything allowed by types within the System.Linq.Expressions namespace and, by extension, anything that can be represented in a lambda function. This is a superset of the expressions that are allowed by the Entity Framework, which are by definition restricted to operations allowed on the database, and supported by ObjectQuery<T>.

In the Entity Framework, both operators and expressions are represented by a single type hierarchy, which are then placed in a command tree. The command tree is used by the Entity Framework to execute the query. If the LINQ query cannot be expressed as a command tree, an exception will be thrown when the query is being converted. The conversion of LINQ to Entities queries involves two sub-conversions: the conversion of the standard query operators, and the conversion of the expressions.

There are a number of LINQ standard query operators that do not have a valid translation in LINQ to Entities. Attempts to use these operators will result in an exception at query translation time. For a list of supported LINQ to Entities operators, see Supported and Unsupported LINQ Methods (LINQ to Entities).

For more information about using the standard query operators in LINQ to Entities, see Standard Query Operators in LINQ to Entities Queries.

In general, expressions in LINQ to Entities are evaluated on the server, so the behavior of the expression should not be expected to follow CLR semantics. For more information, see Expressions in LINQ to Entities Queries.

For information about how CLR method calls are mapped to canonical functions in the data source, see CLR Method to Canonical Function Mapping.

For information about how to call canonical, database, and custom functions from within LINQ to Entities queries, see Calling Functions in LINQ to Entities Queries.

Query Execution

After the LINQ query is created by the user, it is converted to a representation that is compatible with the Entity Framework (in the form of command trees), which is then executed against the data source. At query execution time, all query expressions (or components of the query) are evaluated on the client or on the server. This includes expressions that are used in result materialization or entity projections. For more information, see Query Execution. For information on how to improve performance by compiling a query once and then executing it several times with different parameters, see Compiled Queries (LINQ to Entities).

Materialization

Materialization is the process of returning query results back to the client as CLR types. In LINQ to Entities, query results data records are never returned; there is always a backing CLR type, defined by the user or by the Entity Framework, or generated by the compiler (anonymous types). All object materialization is performed by the Entity Framework. Any errors that result from an inability to map between the Entity Framework and the CLR will cause exceptions to be thrown during object materialization.

Query results are usually returned as one of the following:

- A collection of zero or more typed entity objects or a projection of complex types defined in the conceptual model.
- CLR types that are supported by the Entity Framework.
- Inline collections.
- Anonymous types.

For more information, see Query Results.

In This Section

Queries in LINQ to Entities

Expressions in LINQ to Entities Queries

Calling Functions in LINQ to Entities Queries

Compiled Queries (LINQ to Entities)

Query Execution

Query Results

Standard Query Operators in LINQ to Entities Queries

CLR Method to Canonical Function Mapping

Supported and Unsupported LINQ Methods (LINQ to Entities)

Known Issues and Considerations in LINQ to Entities

See also

- Known Issues and Considerations in LINQ to Entities
- Language-Integrated Query (LINQ) - C#
- Language-Integrated Query (LINQ) - Visual Basic
- LINQ and ADO.NET
- ADO.NET Entity Framework

Queries in LINQ to Entities

11/8/2022 • 3 minutes to read • Edit Online

A query is an expression that retrieves data from a data source. Queries are usually expressed in a specialized query language, such as SQL for relational databases and XQuery for XML. Therefore, developers have had to learn a new query language for each type of data source or data format that they query. Language-Integrated Query (LINQ) offers a simpler, consistent model for working with data across various kinds of data sources and formats. In a LINQ query, you always work with programming objects.

A LINQ query operation consists of three actions: obtain the data source or sources, create the query, and execute the query.

Data sources that implement the IEnumerable<T> generic interface or the IQueryable<T> generic interface can be queried through LINQ. Instances of the generic ObjectQuery<T> class, which implements the generic IQueryable<T> interface, serve as the data source for LINQ to Entities queries. The ObjectQuery<T> generic class represents a query that returns a collection of zero or more typed objects. You can also let the compiler infer the type of an entity by using the C# keyword `var` (Dim in Visual Basic).

In the query, you specify exactly the information that you want to retrieve from the data source. A query can also specify how that information should be sorted, grouped, and shaped before it is returned. In LINQ, a query is stored in a variable. If the query returns a sequence of values, the query variable itself must be a queryable type. This query variable takes no action and returns no data; it only stores the query information. After you create a query you must execute that query to retrieve any data.

Query Syntax

LINQ to Entities queries can be composed in two different syntaxes: query expression syntax and method-based query syntax. Query expression syntax is new in C# 3.0 and Visual Basic 9.0, and it consists of a set of clauses written in a declarative syntax similar to Transact-SQL or XQuery. However, the .NET Framework common language runtime (CLR) cannot read the query expression syntax itself. Therefore, at compile time, query expressions are translated to something that the CLR does understand: method calls. These methods are known as the *standard query operators*. As a developer, you have the option of calling them directly by using method syntax, instead of using query syntax. For more information, see Query Syntax and Method Syntax in LINQ.

Query Expression Syntax

Query expressions are a declarative query syntax. This syntax enables a developer to write queries in a high-level language that is formatted similar to Transact-SQL. By using query expression syntax, you can perform even complex filtering, ordering, and grouping operations on data sources with minimal code. For more information, see Basic Query Operations (Visual Basic). For examples that demonstrate how to use the query expression syntax, see the following topics:

- Query Expression Syntax Examples: Projection
- Query Expression Syntax Examples: Filtering
- Query Expression Syntax Examples: Ordering
- Query Expression Syntax Examples: Aggregate Operators
- Query Expression Syntax Examples: Partitioning
- Query Expression Syntax Examples: Join Operators

- Query Expression Syntax Examples: Element Operators
- Query Expression Syntax Examples: Grouping
- Query Expression Syntax Examples: Navigating Relationships

Method-Based Query Syntax

Another way to compose LINQ to Entities queries is by using method-based queries. The method-based query syntax is a sequence of direct method calls to LINQ operator methods, passing lambda expressions as the parameters. For more information, see Lambda Expressions. For examples that demonstrate how to use method-based syntax, see the following topics:

- Method-Based Query Syntax Examples: Projection
- Method-Based Query Syntax Examples: Filtering
- Method-Based Query Syntax Examples: Ordering
- Method-Based Query Syntax Examples: Aggregate Operators
- Method-Based Query Syntax Examples: Partitioning
- Method-Based Query Syntax Examples: Conversion
- Method-Based Query Syntax Examples: Join Operators
- Method-Based Query Syntax Examples: Element Operators
- Method-Based Query Syntax Examples: Grouping
- Method-Based Query Syntax Examples: Navigating Relationships

See also

- LINQ to Entities
- Getting Started with LINQ in C#
- Getting Started with LINQ in Visual Basic
- EF Merge Options and Compiled Queries

Method-Based Query Syntax Examples: Projection

11/8/2022 • 3 minutes to read • Edit Online

The examples in this topic demonstrate how to use the Select and SelectMany methods to query the AdventureWorks Sales Model using method-based query syntax. The AdventureWorks Sales Model used in these examples is built from the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

```
Option Explicit On
Option Strict On
Imports System.Data.Objects
Imports System.Globalization
```

Select

Example

The following example uses the Select method to project the `Product.Name` and `Product.ProductID` properties into a sequence of anonymous types.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query = context.Products
        .Select(product => new
        {
            ProductId = product.ProductID,
            ProductName = product.Name
        });

    Console.WriteLine("Product Info:");
    foreach (var productInfo in query)
    {
        Console.WriteLine("Product Id: {0} Product name: {1} ",
            productInfo.ProductId, productInfo.ProductName);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim query = context.Products _
    .Select(Function(prod) New With _
    { _
        .ProductName = prod.Name, _
        .ProductId = prod.ProductID _
    })

    Console.WriteLine("Product Info:")
    For Each productInfo In query
        Console.WriteLine("Product Id: {0} Product name: {1} ", _
                productInfo.ProductId, productInfo.ProductName)
    Next
End Using
```

Example

The following example uses the Select method to return a sequence of only product names.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<string> productNames = context.Products
        .Select(p => p.Name);

    Console.WriteLine("Product Names:");
    foreach (String productName in productNames)
    {
        Console.WriteLine(productName);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim productNames = context.Products _
    .Select(Function(p) p.Name())

    Console.WriteLine("Product Names:")
    For Each productName In productNames
        Console.WriteLine(productName)
    Next
End Using
```

SelectMany

Example

The following example uses the SelectMany method to select all orders where `TotalDue` is less than 500.00.

```
decimal totalDue = 500.00M;
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
    contacts.SelectMany(
        contact => orders.Where(order =>
            (contact.ContactID == order.Contact.ContactID)
                && order.TotalDue < totalDue)
            .Select(order => new
            {
                ContactID = contact.ContactID,
                LastName = contact.LastName,
                FirstName = contact.FirstName,
                OrderID = order.SalesOrderID,
                Total = order.TotalDue
            }));

    foreach (var smallOrder in query)
    {
        Console.WriteLine("Contact ID: {0} Name: {1}, {2} Order ID: {3} Total Due: ${4} ",
            smallOrder.ContactID, smallOrder.LastName, smallOrder.FirstName,
            smallOrder.OrderID, smallOrder.Total);
    }
}
```

```
Dim totalDue = 500D
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = contacts.SelectMany( _
        Function(contact) orders.Where(Function(order) _
            (contact.ContactID = order.Contact.ContactID) _
                    And order.TotalDue < totalDue) _
            .Select(Function(order) New With _
            { _
                .ContactID = contact.ContactID, _
                .LastName = contact.LastName, _
                .FirstName = contact.FirstName, _
                .OrderID = order.SalesOrderID, _
                .Total = order.TotalDue _
            }) _
    )

    For Each smallOrder In query
        Console.WriteLine("Contact ID: {0} Name: {1}, {2} Order ID: {3} Total Due: ${4} ", _
            smallOrder.ContactID, smallOrder.LastName, smallOrder.FirstName, _
            smallOrder.OrderID, smallOrder.Total)
    Next
End Using
```

Example

The following example uses the SelectMany method to select all orders where the order was made on October 1, 2002 or later.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
    contacts.SelectMany(
        contact => orders.Where(order =>
            (contact.ContactID == order.Contact.ContactID)
                && order.OrderDate >= new DateTime(2002, 10, 1))
            .Select(order => new
            {
                ContactID = contact.ContactID,
                LastName = contact.LastName,
                FirstName = contact.FirstName,
                OrderID = order.SalesOrderID,
                OrderDate = order.OrderDate
            }));

    foreach (var order in query)
    {
        Console.WriteLine("Contact ID: {0} Name: {1}, {2} Order ID: {3} Order date: {4:d} ",
            order.ContactID, order.LastName, order.FirstName,
            order.OrderID, order.OrderDate);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = contacts.SelectMany( _
        Function(contact) orders.Where(Function(order) _
            (contact.ContactID = order.Contact.ContactID) _
                    And order.OrderDate >= New DateTime(2002, 10, 1)) _
            .Select(Function(order) New With _
            { _
                .ContactID = contact.ContactID, _
                .LastName = contact.LastName, _
                .FirstName = contact.FirstName, _
                .OrderID = order.SalesOrderID, _
                .OrderDate = order.OrderDate _
            }) _
    )

    For Each order In query
        Console.WriteLine("Contact ID: {0} Name: {1}, {2} Order ID: {3} Order date: {4:d} ", _
            order.ContactID, order.LastName, order.FirstName, _
            order.OrderID, order.OrderDate)
    Next
End Using
```

See also

- Queries in LINQ to Entities

Method-Based Query Syntax Examples: Filtering

11/8/2022 • 4 minutes to read • Edit Online

The examples in this topic demonstrate how to use the `Where` and `Where…Contains` methods to query the AdventureWorks Sales Model using method-based query syntax. Note, Where… `Contains` cannot be used as a part of a compiled query.

The AdventureWorks Sales model used in these examples is built from the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

```
Option Explicit On
Option Strict On
Imports System.Data.Objects
Imports System.Globalization
```

Where

Example

The following example returns all online orders.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var onlineOrders = context.SalesOrderHeaders
        .Where(order => order.OnlineOrderFlag == true)
        .Select(s => new { s.SalesOrderID, s.OrderDate, s.SalesOrderNumber });

    foreach (var onlineOrder in onlineOrders)
    {
        Console.WriteLine("Order ID: {0} Order date: {1:d} Order number: {2}",
            onlineOrder.SalesOrderID,
            onlineOrder.OrderDate,
            onlineOrder.SalesOrderNumber);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim onlineOrders = context.SalesOrderHeaders _
        .Where(Function(order) order.OnlineOrderFlag = True) _
        .Select(Function(order) New With { _
           .SalesOrderID = order.SalesOrderID, _
           .OrderDate = order.OrderDate, _
           .SalesOrderNumber = order.SalesOrderNumber _
        })

    For Each onlineOrder In onlineOrders
        Console.WriteLine("Order ID: {0} Order date: {1:d} Order number: {2}", _
                onlineOrder.SalesOrderID, _
                onlineOrder.OrderDate, _
                onlineOrder.SalesOrderNumber)
    Next
End Using
```

Example

The following example returns the orders where the order quantity is greater than 2 and less than 6.

```
int orderQtyMin = 2;
int orderQtyMax = 6;
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query = context.SalesOrderDetails
        .Where(order => order.OrderQty > orderQtyMin && order.OrderQty < orderQtyMax)
        .Select(s => new { s.SalesOrderID, s.OrderQty });

    foreach (var order in query)
    {
        Console.WriteLine("Order ID: {0} Order quantity: {1}",
            order.SalesOrderID, order.OrderQty);
    }
}
```

```
Dim orderQtyMin = 2
Dim orderQtyMax = 6
Using context As New AdventureWorksEntities
    Dim query = context.SalesOrderDetails _
        .Where(Function(order) order.OrderQty > orderQtyMin And order.OrderQty < orderQtyMax) _
        .Select(Function(order) New With { _
            .SalesOrderID = order.SalesOrderID, _
            .OrderQty = order.OrderQty _
        })

    For Each order In query
        Console.WriteLine("Order ID: {0} Order quantity: {1}", _
                order.SalesOrderID, order.OrderQty)
    Next
End Using
```

Example

The following example returns all red colored products.

```
String color = "Red";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query = context.Products
        .Where(product => product.Color == color)
        .Select(p => new { p.Name, p.ProductNumber, p.ListPrice });

    foreach (var product in query)
    {
        Console.WriteLine("Name: {0}", product.Name);
        Console.WriteLine("Product number: {0}", product.ProductNumber);
        Console.WriteLine("List price: ${0}", product.ListPrice);
        Console.WriteLine("");
    }
}
```

```
Dim color = "Red"
Using context As New AdventureWorksEntities
    Dim query = context.Products _
        .Where(Function(product) product.Color = color) _
        .Select(Function(product) New With { _
            .Name = product.Name, _
            .ProductNumber = product.ProductNumber, _
            .ListPrice = product.ListPrice _
        })

    For Each product In query
        Console.WriteLine("Name: {0}", product.Name)
        Console.WriteLine("Product number: {0}", product.ProductNumber)
        Console.WriteLine("List price: ${0}", product.ListPrice)
        Console.WriteLine("")
    Next
End Using
```

Example

The following example uses the `Where` method to find orders that were made after December 1, 2003 and then uses the `order.SalesOrderDetail` navigation property to get the details for each order.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<SalesOrderHeader> query = context.SalesOrderHeaders
        .Where(order => order.OrderDate >= new DateTime(2003, 12, 1));

    Console.WriteLine("Orders that were made after December 1, 2003:");
    foreach (SalesOrderHeader order in query)
    {
        Console.WriteLine("OrderID {0} Order date: {1:d} ",
            order.SalesOrderID, order.OrderDate);
        foreach (SalesOrderDetail orderDetail in order.SalesOrderDetails)
        {
            Console.WriteLine("  Product ID: {0} Unit Price {1}",
                orderDetail.ProductID, orderDetail.UnitPrice);
        }
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim query = context.SalesOrderHeaders _
        .Where(Function(order) order.OrderDate >= New DateTime(2003, 12, 1)) _
        .Select(Function(order) order)

    Console.WriteLine("Orders that were made after December 1, 2003:")
    For Each order In query
        Console.WriteLine("OrderID {0} Order date: {1:d} ", _
                order.SalesOrderID, order.OrderDate)
        For Each orderDetail In order.SalesOrderDetails
            Console.WriteLine("  Product ID: {0} Unit Price {1}", _
                orderDetail.ProductID, orderDetail.UnitPrice)
        Next
    Next
End Using
```

Where...Contains

Example

The following example uses an array as part of a `Where...Contains` clause to find all products that have a `ProductModelID` that matches a value in the array.

```
using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
{
    int?[] productModelIds = { 19, 26, 118 };
    var products = AWEntities.Products.
        Where(p => productModelIds.Contains(p.ProductModelID));

    foreach (var product in products)
    {
        Console.WriteLine("{0}: {1}", product.ProductModelID, product.ProductID);
    }
}
```

```
Using AWEntities As New AdventureWorksEntities()
    Dim productModelIds As System.Nullable(Of Integer)() = {19, 26, 118}
    Dim products = AWEntities.Products. _
        Where(Function(p) productModelIds.Contains(p.ProductModelID))
    For Each product In products
        Console.WriteLine("{0}: {1}", product.ProductModelID, product.ProductID)
    Next
End Using
```

NOTE

As part of the predicate in a `Where...Contains` clause, you can use an Array, a List<T>, or a collection of any type that implements the IEnumerable<T> interface. You can also declare and initialize a collection within a LINQ to Entities query. See the next example for more information.

Example

The following example declares and initializes arrays in a `Where...Contains` clause to find all products that have a `ProductModelID` or a `Size` that matches a value in the arrays.

```
using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
{
    var products = AWEntities.Products.
        Where(p => (new int?[] { 19, 26, 18 }).Contains(p.ProductModelID) ||
                   (new string[] { "L", "XL" }).Contains(p.Size));

    foreach (var product in products)
    {
        Console.WriteLine("{0}: {1}, {2}", product.ProductID,
                                           product.ProductModelID,
                                           product.Size);
    }
}
```

```
Using AWEntities As New AdventureWorksEntities()
    Dim products = AWEntities.Products. _
        Where(Function(p) (New System.Nullable(Of Integer)() {19, 26, 18}). _
                          Contains(p.ProductModelID) _
              OrElse _
                         (New String() {"L", "XL"}).Contains(p.Size))

    For Each product In products
        Console.WriteLine("{0}: {1}, {2}", product.ProductID, _
                                           product.ProductModelID, _
                                           product.Size)
    Next
End Using
```

See also

- Queries in LINQ to Entities

Method-Based Query Syntax Examples: Ordering

11/8/2022 • 2 minutes to read • Edit Online

The examples in this topic demonstrate how to use the ThenBy method to query the AdventureWorks Sales Model using method-based query syntax. The AdventureWorks Sales Model used in these examples is built from the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

```
Option Explicit On
Option Strict On
Imports System.Data.Objects
Imports System.Globalization
```

ThenBy

Example

The following example in method-based query syntax uses OrderBy and ThenBy to return a list of contacts ordered by last name and then by first name.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Contact> sortedContacts = context.Contacts
        .OrderBy(c => c.LastName)
        .ThenBy(c => c.FirstName);

    Console.WriteLine("The list of contacts sorted by last name then by first name:");
    foreach (Contact sortedContact in sortedContacts)
    {
        Console.WriteLine(sortedContact.LastName + ", " + sortedContact.FirstName);
    }
}
```

```
Using context As New AdventureWorksEntities

    Dim sortedContacts = context.Contacts _
    .OrderBy(Function(c) c.LastName) _
    .ThenBy(Function(c) c.FirstName) _
    .Select(Function(c) c)

    Console.WriteLine("The list of contacts sorted by last name then by first name:")
    For Each sortedContact As Contact In sortedContacts
        Console.WriteLine(sortedContact.LastName + ", " + sortedContact.FirstName)
    Next
End Using
```

ThenByDescending

Example

The following example uses the OrderBy and ThenByDescending methods to first sort by list price, and then perform a descending sort of the product names.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IOrderedQueryable<Product> query = context.Products
        .OrderBy(product => product.ListPrice)
        .ThenByDescending(product => product.Name);

    foreach (Product product in query)
    {
        Console.WriteLine("Product ID: {0} Product Name: {1} List Price {2}",
            product.ProductID,
            product.Name,
            product.ListPrice);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim products As ObjectSet(Of Product) = context.Products

    Dim query As IOrderedQueryable(Of Product) = products _
    .OrderBy(Function(prod As Product) prod.ListPrice) _
    .ThenByDescending(Function(prod As Product) prod.Name)

    For Each prod As Product In query
        Console.WriteLine("Product ID: {0} Product Name: {1} List Price {2}", _
            prod.ProductID, _
            prod.Name, _
            prod.ListPrice)
    Next
End Using
```

See also

- Queries in LINQ to Entities

Method-Based Query Syntax Examples: Aggregate Operators

11/8/2022 • 10 minutes to read • Edit Online

The examples in this topic demonstrate how to use the Aggregate, Average, Count, LongCount, Max, Min, and Sum methods to query the AdventureWorks Sales Model using method-based query syntax. The AdventureWorks Sales Model used in these examples is built from the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

```
Option Explicit On
Option Strict On
Imports System.Data.Objects
Imports System.Globalization
```

Average

Example

The following example uses the Average method to find the average list price of the products.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Product> products = context.Products;

    Decimal averageListPrice =
        products.Average(product => product.ListPrice);

    Console.WriteLine("The average list price of all the products is ${0}",
        averageListPrice);
}
```

```
Using context As New AdventureWorksEntities
    Dim products As ObjectSet(Of Product) = context.Products

    Dim averageListPrice As Decimal = _
        products.Average(Function(prod) prod.ListPrice)

    Console.WriteLine("The average list price of all the products is ${0}", _
            averageListPrice)
End Using
```

Example

The following example uses the Average method to find the average list price of the products of each style.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Product> products = context.Products;

    var query = from product in products
                group product by product.Style into g
                select new
                {
                    Style = g.Key,
                    AverageListPrice =
                        g.Average(product => product.ListPrice)
                };

    foreach (var product in query)
    {
        Console.WriteLine("Product style: {0} Average list price: {1}",
            product.Style, product.AverageListPrice);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim products As ObjectSet(Of Product) = context.Products

    Dim query = _
        From prod In products _
        Let styl = prod.Style _
        Group prod By styl Into g = Group _
        Select New With _
        { _
            .Style = styl, _
            .AverageListPrice = g.Average(Function(p) p.ListPrice) _
        }

    For Each prod In query
        Console.WriteLine("Product style: {0} Average list price: {1}", _
            prod.Style, prod.AverageListPrice)
    Next
End Using
```

Example

The following example uses the Average method to find the average total due.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    Decimal averageTotalDue = orders.Average(order => order.TotalDue);
    Console.WriteLine("The average TotalDue is {0}.", averageTotalDue);
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim averageTotalDue As Decimal = _
        orders.Average(Function(ord) ord.TotalDue)

    Console.WriteLine("The average TotalDue is {0}.", averageTotalDue)
End Using
```

Example

The following example uses the Average method to get the average total due for each contact ID.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from order in orders
        group order by order.Contact.ContactID into g
        select new
        {
            Category = g.Key,
            averageTotalDue = g.Average(order => order.TotalDue)
        };

    foreach (var order in query)
    {
        Console.WriteLine("ContactID = {0} \t Average TotalDue = {1}",
            order.Category, order.averageTotalDue);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From ord In orders _
        Let contID = ord.Contact.ContactID _
        Group ord By contID Into g = Group _
        Select New With _
        { _
            .Category = contID, _
            .averageTotalDue = _
                g.Average(Function(ord) ord.TotalDue) _
         }

    For Each ord In query
        Console.WriteLine("ContactID = {0} " & vbTab & _
            " Average TotalDue = {1}", _
            ord.Category, ord.averageTotalDue)
    Next
End Using
```

Example

The following example uses the Average method to get the orders with the average total due for each contact.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from order in orders
        group order by order.Contact.ContactID into g
        let averageTotalDue = g.Average(order => order.TotalDue)
        select new
        {
            Category = g.Key,
            CheapestProducts =
                g.Where(order => order.TotalDue == averageTotalDue)
        };

    foreach (var orderGroup in query)
    {
        Console.WriteLine("ContactID: {0}", orderGroup.Category);
        foreach (var order in orderGroup.CheapestProducts)
        {
            Console.WriteLine("Average total due for SalesOrderID {1} is: {0}",
                order.TotalDue, order.SalesOrderID);
        }
        Console.Write("\n");
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From ord In orders _
        Let contID = ord.Contact.ContactID _
        Group ord By contID Into g = Group _
        Let averageTotalDue = g.Average(Function(ord) ord.TotalDue) _
        Select New With _
        { _
            .Category = contID, _
            .CheapestProducts = _
                g.Where(Function(ord) ord.TotalDue = averageTotalDue) _
         }

    For Each orderGroup In query
        Console.WriteLine("ContactID: {0}", orderGroup.Category)
        For Each ord In orderGroup.CheapestProducts
            Console.WriteLine("Average total due for SalesOrderID {1} is: {0}", _
                ord.TotalDue, ord.SalesOrderID)
        Next
        Console.Write(vbNewLine)
    Next
End Using
```

Count

Example

The following example uses the Count method to return the number of products in the Product table.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Product> products = context.Products;

    int numProducts = products.Count();

    Console.WriteLine("There are {0} products.", numProducts);
}
```

```
Using context As New AdventureWorksEntities
    Dim products As ObjectSet(Of Product) = context.Products

    Dim numProducts As Integer = products.Count()

    Console.WriteLine("There are {0} products.", numProducts)
End Using
```

Example

The following example uses the Count method to return a list of contact IDs and how many orders each has.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;

    //Can't find field SalesOrderContact
    var query =
        from contact in contacts
        select new
        {
            CustomerID = contact.ContactID,
            OrderCount = contact.SalesOrderHeaders.Count()
        };

    foreach (var contact in query)
    {
        Console.WriteLine("CustomerID = {0} \t OrderCount = {1}",
            contact.CustomerID,
            contact.OrderCount);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts

    Dim query = _
        From cont In contacts _
        Select New With _
        { _
            .CustomerID = cont.ContactID, _
            .OrderCount = cont.SalesOrderHeaders.Count() _
         }

    For Each cont In query
        Console.WriteLine("CustomerID = {0}   OrderCount = {1}", _
               cont.CustomerID, cont.OrderCount)
    Next
End Using
```

Example

The following example groups products by color and uses the Count method to return the number of products

in each color group.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Product> products = context.Products;

    var query =
        from product in products
        group product by product.Color into g
        select new { Color = g.Key, ProductCount = g.Count() };

    foreach (var product in query)
    {
        Console.WriteLine("Color = {0} \t ProductCount = {1}",
            product.Color,
            product.ProductCount);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim products As ObjectSet(Of Product) = context.Products

    Dim query = _
        From prod In products _
        Let pc = prod.Color _
        Group prod By pc Into g = Group _
        Select New With {.Color = pc, .ProductCount = g.Count()}

    For Each prod In query
        Console.WriteLine("Color = {0} " & vbTab & " ProductCount = {1}", _
            prod.Color, prod.ProductCount)
    Next
End Using
```

LongCount

Example

The following example gets the contact count as a long integer.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;

    long numberOfContacts = contacts.LongCount();
    Console.WriteLine("There are {0} Contacts", numberOfContacts);
}
```

```
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts

    Dim numberOfContacts As Long = contacts.LongCount()

    Console.WriteLine("There are {0} Contacts", numberOfContacts)
End Using
```

Max

Example

The following example uses the Max method to get the largest total due.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    Decimal maxTotalDue = orders.Max(w => w.TotalDue);
    Console.WriteLine("The maximum TotalDue is {0}.",
        maxTotalDue);
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim maxTotalDue As Decimal = _
        orders.Max(Function(ord) ord.TotalDue)

    Console.WriteLine("The maximum TotalDue is {0}.", maxTotalDue)
End Using
```

Example

The following example uses the Max method to get the largest total due for each contact ID.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from order in orders
        group order by order.Contact.ContactID into g
        select new
        {
            Category = g.Key,
            maxTotalDue =
                g.Max(order => order.TotalDue)
        };

    foreach (var order in query)
    {
        Console.WriteLine("ContactID = {0} \t Maximum TotalDue = {1}",
            order.Category, order.maxTotalDue);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From ord In orders _
        Let contID = ord.Contact.ContactID _
        Group ord By contID Into g = Group _
        Select New With _
        { _
            .Category = contID, _
            .MaxTotalDue = _
                g.Max(Function(ord) ord.TotalDue) _
         }

    For Each ord In query
        Console.WriteLine("ContactID = {0} " & vbTab & _
            " Maximum TotalDue = {1}", _
            ord.Category, ord.MaxTotalDue)
    Next
End Using
```

Example

The following example uses the Max method to get the orders with the largest total due for each contact ID.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from order in orders
        group order by order.Contact.ContactID into g
        let maxTotalDue = g.Max(order => order.TotalDue)
        select new
        {
            Category = g.Key,
            CheapestProducts =
                g.Where(order => order.TotalDue == maxTotalDue)
        };

    foreach (var orderGroup in query)
    {
        Console.WriteLine("ContactID: {0}", orderGroup.Category);
        foreach (var order in orderGroup.CheapestProducts)
        {
            Console.WriteLine("MaxTotalDue {0} for SalesOrderID {1}: ",
                order.TotalDue,
                order.SalesOrderID);
        }
        Console.Write("\n");
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From ord In orders _
        Let contID = ord.Contact.ContactID _
        Group ord By contID Into g = Group _
        Let maxTotalDue = g.Max(Function(ord) ord.TotalDue) _
        Select New With _
        { _
            .Category = contID, _
            .CheapestProducts = _
                g.Where(Function(ord) ord.TotalDue = maxTotalDue) _
         }

    For Each orderGroup In query
        Console.WriteLine("ContactID: {0}", orderGroup.Category)
        For Each ord In orderGroup.CheapestProducts
            Console.WriteLine("MaxTotalDue {0} for SalesOrderID {1}: ", _
                ord.TotalDue, ord.SalesOrderID)
        Next
        Console.Write(vbNewLine)
    Next
End Using
```

Min

Example

The following example uses the Min method to get the smallest total due.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    Decimal smallestTotalDue = orders.Min(totalDue => totalDue.TotalDue);
    Console.WriteLine("The smallest TotalDue is {0}.",
        smallestTotalDue);
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim smallestTotalDue As Decimal = _
        orders.Min(Function(totDue) totDue.TotalDue)

    Console.WriteLine("The smallest TotalDue is {0}.", _
        smallestTotalDue)
End Using
```

Example

The following example uses the Min method to get the smallest total due for each contact ID.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from order in orders
        group order by order.Contact.ContactID into g
        select new
        {
            Category = g.Key,
            smallestTotalDue =
                g.Min(order => order.TotalDue)
        };

    foreach (var order in query)
    {
        Console.WriteLine("ContactID = {0} \t Minimum TotalDue = {1}",
            order.Category, order.smallestTotalDue);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From ord In orders _
        Let contID = ord.Contact.ContactID _
        Group ord By contID Into g = Group _
        Select New With _
        { _
            .Category = contID, _
            .smallestTotalDue = _
                g.Min(Function(o) o.TotalDue) _
         }

    For Each ord In query
        Console.WriteLine("ContactID = {0} " & vbTab & _
            " Minimum TotalDue = {1}", ord.Category, ord.smallestTotalDue)
    Next
End Using
```

Example

The following example uses the Min method to get the orders with the smallest total due for each contact.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from order in orders
        group order by order.Contact.ContactID into g
        let minTotalDue = g.Min(order => order.TotalDue)
        select new
        {
            Category = g.Key,
            smallestTotalDue =
                g.Where(order => order.TotalDue == minTotalDue)
        };

    foreach (var orderGroup in query)
    {
        Console.WriteLine("ContactID: {0}", orderGroup.Category);
        foreach (var order in orderGroup.smallestTotalDue)
        {
            Console.WriteLine("Mininum TotalDue {0} for SalesOrderID {1}: ",
                order.TotalDue,
                order.SalesOrderID);
        }
        Console.Write("\n");
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From ord In orders _
        Let contID = ord.Contact.ContactID _
        Group ord By contID Into g = Group _
        Let minTotalDue = g.Min(Function(o) o.TotalDue) _
        Select New With _
        { _
            .Category = contID, _
            .smallestTotalDue = _
                g.Where(Function(o) o.TotalDue = minTotalDue) _
         }

    For Each orderGroup In query
        Console.WriteLine("ContactID: {0}", orderGroup.Category)
        For Each ord In orderGroup.smallestTotalDue
            Console.WriteLine("Mininum TotalDue {0} for SalesOrderID {1}: ", _
                ord.TotalDue, ord.SalesOrderID)
        Next
        Console.Write(vbNewLine)
    Next
End Using
```

Sum

Example

The following example uses the Sum method to get the total number of order quantities in the SalesOrderDetail table.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderDetail> orders = context.SalesOrderDetails;

    double totalOrderQty = orders.Sum(o => o.OrderQty);
    Console.WriteLine("There are a total of {0} OrderQty.",
        totalOrderQty);
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderDetail) = context.SalesOrderDetails

    Dim totalOrderQty As Double = orders.Sum(Function(o) o.OrderQty)

    Console.WriteLine("There are a total of {0} OrderQty.", _
        totalOrderQty)
End Using
```

Example

The following example uses the Sum method to get the total due for each contact ID.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from order in orders
        group order by order.Contact.ContactID into g
        select new
        {
            Category = g.Key,
            TotalDue = g.Sum(order => order.TotalDue)
        };

    foreach (var order in query)
    {
        Console.WriteLine("ContactID = {0} \t TotalDue sum = {1}",
            order.Category, order.TotalDue);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From ord In orders _
        Let contID = ord.Contact.ContactID _
        Group ord By contID Into g = Group _
        Select New With _
        { _
            .Category = contID, _
            .TotalDue = g.Sum(Function(o) o.TotalDue) _
        }

    For Each ord In query
        Console.WriteLine("ContactID = {0} " & vbTab & _
            " TotalDue sum = {1}", ord.Category, ord.TotalDue)
    Next
End Using
```

See also

- Queries in LINQ to Entities

Method-Based Query Syntax Examples: Partitioning

11/8/2022 • 3 minutes to read • Edit Online

The examples in this topic demonstrate how to use the Skip, and Take methods to query the AdventureWorks Sales Model using query expression syntax. The AdventureWorks Sales Model used in these examples is built from the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

```
Option Explicit On
Option Strict On
Imports System.Data.Objects
Imports System.Globalization
```

Skip

Example

The following example uses the Skip method to get all but the first three contacts of the Contact table.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    // LINQ to Entities only supports Skip on ordered collections.
    IOrderedQueryable<Product> products = context.Products
            .OrderBy(p => p.ListPrice);

    IQueryable<Product> allButFirst3Products = products.Skip(3);

    Console.WriteLine("All but first 3 products:");
    foreach (Product product in allButFirst3Products)
    {
        Console.WriteLine("Name: {0} \t ID: {1}",
            product.Name,
            product.ProductID);
    }
}
```

```
Using context As New AdventureWorksEntities
    'LINQ to Entities only supports Skip on ordered collections.
    Dim products As IOrderedQueryable(Of Product) = _
            context.Products.OrderBy(Function(p) p.ListPrice)

    Dim allButFirst3Products As IQueryable(Of Product) = products.Skip(3)

    Console.WriteLine("All but first 3 products:")
    For Each product As Product In allButFirst3Products
        Console.WriteLine("Name: {0} \t ID: {1}", _
                product.Name, _
                product.ProductID)
    Next
End Using
```

Example

The following example uses the Skip method to get all but the first two addresses in Seattle.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Address> addresses = context.Addresses;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    //LINQ to Entities only supports Skip on ordered collections.
    var query = (
        from address in addresses
        from order in orders
        where address.AddressID == order.Address.AddressID
             && address.City == "Seattle"
        orderby order.SalesOrderID
        select new
        {
            City = address.City,
            OrderID = order.SalesOrderID,
            OrderDate = order.OrderDate
        }).Skip(2);

    Console.WriteLine("All but first 2 orders in Seattle:");
    foreach (var order in query)
    {
        Console.WriteLine("City: {0} Order ID: {1} Total Due: {2:d}",
            order.City, order.OrderID, order.OrderDate);
    }
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders
    Dim addresses As ObjectSet(Of Address) = context.Addresses

    'LINQ to Entities only supports Skip on ordered collections.
    Dim query = ( _
        From address In addresses _
        From order In orders _
        Where address.AddressID = order.Address.AddressID _
                 And address.City = "Seattle" _
        Order By order.SalesOrderID _
        Select New With _
        { _
            .City = address.City, _
            .OrderID = order.SalesOrderID, _
            .OrderDate = order.OrderDate _
        }).Skip(2)

    Console.WriteLine("All but first 2 orders in Seattle:")
    For Each order In query
        Console.WriteLine("City: {0} Order ID: {1} Total Due: {2:d}", _
            order.City, order.OrderID, order.OrderDate)
    Next
End Using
```

Take

Example

The following example uses the Take method to get only the first five contacts from the Contact table.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Contact> first5Contacts = context.Contacts.Take(5);

    Console.WriteLine("First 5 contacts:");
    foreach (Contact contact in first5Contacts)
    {
        Console.WriteLine("Title = {0} \t FirstName = {1} \t Lastname = {2}",
            contact.Title,
            contact.FirstName,
            contact.LastName);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts

    Dim first5Contacts As IQueryable(Of Contact) = contacts.Take(5)

    Console.WriteLine("First 5 contacts:")
    For Each contact As Contact In first5Contacts
        Console.WriteLine("Title = {0} " & vbTab & " FirstName = {1} " _
            & vbTab & " Lastname = {2}", contact.Title, contact.FirstName, _
            contact.LastName)
    Next
End Using
```

Example

The following example uses the Take method to get the first three addresses in Seattle.

```
String city = "Seattle";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Address> addresses = context.Addresses;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query = (
        from address in addresses
        from order in orders
        where address.AddressID == order.Address.AddressID
             && address.City == city
        select new
        {
            City = address.City,
            OrderID = order.SalesOrderID,
            OrderDate = order.OrderDate
        }).Take(3);
    Console.WriteLine("First 3 orders in Seattle:");
    foreach (var order in query)
    {
        Console.WriteLine("City: {0} Order ID: {1} Total Due: {2:d}",
            order.City, order.OrderID, order.OrderDate);
    }
}
```

```
Dim city = "Seattle"
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders
    Dim addresses As ObjectSet(Of Address) = context.Addresses

    Dim query = ( _
        From address In addresses _
        From order In orders _
        Where address.AddressID = order.Address.AddressID _
                 And address.City = city _
        Select New With _
        { _
            .City = address.City, _
            .OrderID = order.SalesOrderID, _
            .OrderDate = order.OrderDate _
        }).Take(3)

    Console.WriteLine("First 3 orders in Seattle:")
    For Each order In query
        Console.WriteLine("City: {0} Order ID: {1} Total Due: {2:d}", _
            order.City, order.OrderID, order.OrderDate)
    Next
End Using
```

See also

- Queries in LINQ to Entities

Method-Based Query Syntax Examples: Conversion

11/8/2022 • 2 minutes to read • Edit Online

The examples in this topic demonstrate how to use the ToArray, ToDictionary and ToList methods to query the AdventureWorks Sales Model using method-based query syntax. The AdventureWorks Sales Model used in these examples is built from the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

```
Option Explicit On
Option Strict On
Imports System.Data.Objects
Imports System.Globalization
```

ToArray

Example

The following example uses the ToArray method to immediately evaluate a sequence into an array.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Product> products = context.Products;

    Product[] prodArray = (
        from product in products
        orderby product.ListPrice descending
        select product).ToArray();

    Console.WriteLine("Every price from highest to lowest:");
    foreach (Product product in prodArray)
    {
        Console.WriteLine(product.ListPrice);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim products As ObjectSet(Of Product) = context.Products

    Dim prodArray As Product() = ( _
        From product In products _
        Order By product.ListPrice Descending _
        Select product).ToArray()

    Console.WriteLine("The list price from highest to lowest:")
    For Each prod As Product In prodArray
        Console.WriteLine(prod.ListPrice)
    Next
End Using
```

ToDictionary

Example

The following example uses the ToDictionary method to immediately evaluate a sequence and a related key expression into a dictionary.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Product> products = context.Products;

    Dictionary<String, Product> scoreRecordsDict = products.
            ToDictionary(record => record.Name);

    Console.WriteLine("Top Tube's ProductID: {0}",
            scoreRecordsDict["Top Tube"].ProductID);
}
```

```
Using context As New AdventureWorksEntities
    Dim products As ObjectSet(Of Product) = context.Products

    Dim scoreRecordsDict As Dictionary(Of String, Product) = _
        products.ToDictionary(Function(record) record.Name)

    Console.WriteLine("Top Tube's ProductID: {0}", _
        scoreRecordsDict("Top Tube").ProductID)
End Using
```

ToList

Example

The following example uses the ToList method to immediately evaluate a sequence into a List<T>, where `T` is of type DataRow.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Product> products = context.Products;

    List<Product> query =
        (from product in products
         orderby product.Name
         select product).ToList();

    Console.WriteLine("The product list, ordered by product name:");
    foreach (Product product in query)
    {
        Console.WriteLine(product.Name.ToLower(CultureInfo.InvariantCulture));
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim products As ObjectSet(Of Product) = context.Products

    Dim prodList As List(Of Product) = ( _
        From product In products _
        Order By product.Name _
        Select product).ToList()

    Console.WriteLine("The product list, ordered by product name:")
    For Each prod As Product In prodList
        Console.WriteLine(prod.Name.ToLower(CultureInfo.InvariantCulture))
    Next
End Using
```

See also

- Queries in LINQ to Entities

Method-Based Query Syntax Examples: Join Operators

11/8/2022 • 4 minutes to read • Edit Online

The examples in this topic demonstrate how to use the Join and GroupJoin methods to query the AdventureWorks Sales Model using method-based query syntax. The AdventureWorks Sales Model used in these examples is built from the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

```
Option Explicit On
Option Strict On
Imports System.Data.Objects
Imports System.Globalization
```

GroupJoin

Example

The following example performs a GroupJoin over the SalesOrderHeader and SalesOrderDetail tables to find the number of orders per customer. A group join is the equivalent of a left outer join, which returns each element of the first (left) data source, even if no correlated elements are in the other data source.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;
    ObjectSet<SalesOrderDetail> details = context.SalesOrderDetails;

    var query = orders.GroupJoin(details,
        order => order.SalesOrderID,
        detail => detail.SalesOrderID,
        (order, orderGroup) => new
        {
            CustomerID = order.SalesOrderID,
            OrderCount = orderGroup.Count()
        });

    foreach (var order in query)
    {
        Console.WriteLine("CustomerID: {0}  Orders Count: {1}",
            order.CustomerID,
            order.OrderCount);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders
    Dim details As ObjectSet(Of SalesOrderDetail) = context.SalesOrderDetails

    Dim query = orders.GroupJoin(details, _
            Function(order) order.SalesOrderID, _
            Function(detail) detail.SalesOrderID, _
            Function(order, orderGroup) New With _
            { _
                .CustomerID = order.SalesOrderID, _
                .OrderCount = orderGroup.Count() _
            })

    For Each order In query
        Console.WriteLine("CustomerID: {0}  Orders Count: {1}", _
            order.CustomerID, order.OrderCount)
    Next

End Using
```

Example

The following example performs a GroupJoin over the Contact and SalesOrderHeader tables to find the number of orders per contact. The order count and IDs for each contact are displayed.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query = contacts.GroupJoin(orders,
        contact => contact.ContactID,
        order => order.Contact.ContactID,
        (contact, contactGroup) => new
        {
            ContactID = contact.ContactID,
            OrderCount = contactGroup.Count(),
            Orders = contactGroup
        });

    foreach (var group in query)
    {
        Console.WriteLine("ContactID: {0}", group.ContactID);
        Console.WriteLine("Order count: {0}", group.OrderCount);
        foreach (var orderInfo in group.Orders)
        {
            Console.WriteLine("   Sale ID: {0}", orderInfo.SalesOrderID);
        }
        Console.WriteLine("");
    }
    }
```

```
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = contacts.GroupJoin(orders, _
            Function(contact) contact.ContactID, _
            Function(order) order.Contact.ContactID, _
            Function(contact, contactGroup) New With _
            { _
                .ContactID = contact.ContactID, _
                .OrderCount = contactGroup.Count(), _
                .orders = contactGroup.Select(Function(order) order) _
            })

    For Each group In query
        Console.WriteLine("ContactID: {0}", group.ContactID)
        Console.WriteLine("Order count: {0}", group.OrderCount)

        For Each orderInfo In group.orders
            Console.WriteLine("   Sale ID: {0}", orderInfo.SalesOrderID)
        Next

        Console.WriteLine("")
    Next
End Using
```

Join

Example

The following example performs a join over the Contact and SalesOrderHeader tables.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        contacts.Join(
            orders,
            order => order.ContactID,
            contact => contact.Contact.ContactID,
            (contact, order) => new
            {
                ContactID = contact.ContactID,
                SalesOrderID = order.SalesOrderID,
                FirstName = contact.FirstName,
                Lastname = contact.LastName,
                TotalDue = order.TotalDue
            });

    foreach (var contact_order in query)
    {
        Console.WriteLine("ContactID: {0} "
                        + "SalesOrderID: {1} "
                        + "FirstName: {2} "
                        + "Lastname: {3} "
                        + "TotalDue: {4}",
            contact_order.ContactID,
            contact_order.SalesOrderID,
            contact_order.FirstName,
            contact_order.Lastname,
            contact_order.TotalDue);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        contacts.Join( _
            orders, _
            Function(ord) ord.ContactID, _
            Function(cont) cont.Contact.ContactID, _
            Function(cont, ord) New With _
                { _
                    .ContactID = cont.ContactID, _
                    .SalesOrderID = ord.SalesOrderID, _
                    .FirstName = cont.FirstName, _
                    .Lastname = cont.LastName, _
                    .TotalDue = ord.TotalDue _
                })

    For Each contact_order In query
        Console.WriteLine("ContactID: {0} " _
            & "SalesOrderID: {1} " & "FirstName: {2} " _
            & "Lastname: {3} " & "TotalDue: {4}", _
            contact_order.ContactID, _
            contact_order.SalesOrderID, _
            contact_order.FirstName, _
            contact_order.Lastname, _
            contact_order.TotalDue)
    Next
End Using
```

Example

The following example performs a join over the Contact and SalesOrderHeader tables, grouping the results by contact ID.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query = contacts.Join(
        orders,
        order => order.ContactID,
        contact => contact.Contact.ContactID,
        (contact, order) => new
        {
            ContactID = contact.ContactID,
            SalesOrderID = order.SalesOrderID,
            FirstName = contact.FirstName,
            Lastname = contact.LastName,
            TotalDue = order.TotalDue
        })
            .GroupBy(record => record.ContactID);

    foreach (var group in query)
    {
        foreach (var contact_order in group)
        {
            Console.WriteLine("ContactID: {0} "
                            + "SalesOrderID: {1} "
                            + "FirstName: {2} "
                            + "Lastname: {3} "
                            + "TotalDue: {4}",
                contact_order.ContactID,
                contact_order.SalesOrderID,
                contact_order.FirstName,
                contact_order.Lastname,
                contact_order.TotalDue);
        }
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        contacts.Join( _
            orders, _
            Function(ord) ord.ContactID, _
            Function(cont) cont.Contact.ContactID, _
            Function(cont, ord) New With _
                { _
                    .ContactID = cont.ContactID, _
                    .SalesOrderID = ord.SalesOrderID, _
                    .FirstName = cont.FirstName, _
                    .Lastname = cont.LastName, _
                    .TotalDue = ord.TotalDue _
                }) _
                .GroupBy(Function(record) record.ContactID)

    For Each group In query
        For Each contact_order In group
            Console.WriteLine("ContactID: {0} " _
                & "SalesOrderID: {1} " & "FirstName: {2} " _
                & "Lastname: {3} " & "TotalDue: {4}", _
                contact_order.ContactID, _
                contact_order.SalesOrderID, _
                contact_order.FirstName, _
                contact_order.Lastname, _
                contact_order.TotalDue)
        Next
    Next
End Using
```

See also

- Queries in LINQ to Entities

Method-Based Query Syntax Examples: Element Operators

11/8/2022 • 2 minutes to read • Edit Online

The examples in this topic demonstrate how to use the First method to query the AdventureWorks Sales Model using method-based query syntax. The AdventureWorks Sales Model used in these examples is built from the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The example in this topic uses the following `using` / `Imports` statements:

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

```
Option Explicit On
Option Strict On
Imports System.Data.Objects
Imports System.Globalization
```

First

Example

The following example uses the First method to find the first email address that starts with 'caroline'.

```
string name = "caroline";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;

    Contact query = contacts.First(contact =>
        contact.EmailAddress.StartsWith(name));

    Console.WriteLine("An email address starting with 'caroline': {0}",
        query.EmailAddress);
}
```

```
Dim name = "caroline"
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts

    Dim query = contacts.First(Function(cont) _
            cont.EmailAddress.StartsWith(name))

    Console.WriteLine("An email address starting with 'caroline': {0}", _
            query.EmailAddress)
End Using
```

See also

- Queries in LINQ to Entities

Method-Based Query Syntax Examples: Grouping

11/8/2022 • 2 minutes to read • Edit Online

The examples in this topic show you how to use the `GroupBy` method to query the AdventureWorks Sales Model using method-based query syntax. The AdventureWorks Sales Model that is used in these examples is built from the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

```
Option Explicit On
Option Strict On
Imports System.Data.Objects
Imports System.Globalization
```

Example 1

The following example uses the `GroupBy` method to return `Address` objects that are grouped by postal code. The results are projected into an anonymous type.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query = context.Addresses
        .GroupBy( address => address.PostalCode);

    foreach (IGrouping<string, Address> addressGroup in query)
    {
        Console.WriteLine("Postal Code: {0}", addressGroup.Key);
        foreach (Address address in addressGroup)
        {
            Console.WriteLine("\t" + address.AddressLine1 +
                address.AddressLine2);
        }
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim query = context.Addresses _
        .GroupBy(Function(Address) Address.PostalCode) _
        .Select(Function(Address) Address)

    For Each addressGroup As IGrouping(Of String, Address) In query
        Console.WriteLine("Postal Code: {0}", addressGroup.Key)
        For Each address As Address In addressGroup

            Console.WriteLine("   " + address.AddressLine1 + address.AddressLine2)
        Next
    Next
End Using
```

Example 2

The following example uses the `GroupBy` method to return `Contact` objects that are grouped by the first letter of the contact's last name. The results are also sorted by the first letter of the last name and projected into an anonymous type.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query = context.Contacts
        .GroupBy(c => c.LastName.Substring(0,1))
        .OrderBy(c => c.Key);

    foreach (IGrouping<string, Contact> group in query)
    {
        Console.WriteLine("Last names that start with the letter '{0}':",
            group.Key);
        foreach (Contact contact in group)
        {
            Console.WriteLine(contact.LastName);
        }
    }
}
```

```
Using context As New AdventureWorksEntities

    Dim query = context.Contacts _
    .GroupBy(Function(c) c.LastName.Substring(0, 1)) _
    .OrderBy(Function(c) c.Key) _
    .Select(Function(c) c)

    For Each group As IGrouping(Of String, Contact) In query
        Console.WriteLine("Last names that start with the letter '{0}':", group.Key)

        For Each contact As Contact In group

            Console.WriteLine(contact.LastName)
        Next
    Next

End Using
```

Example 3

The following example uses the `GroupBy` method to return `SalesOrderHeader` objects that are grouped by customer ID. The number of sales for each customer is also returned.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query = context.SalesOrderHeaders
        .GroupBy(order => order.CustomerID);

    foreach (IGrouping<int, SalesOrderHeader> group in query)
    {
        Console.WriteLine("Customer ID: {0}", group.Key);
        Console.WriteLine("Order count: {0}", group.Count());

        foreach (SalesOrderHeader sale in group)
        {
            Console.WriteLine("   Sale ID: {0}", sale.SalesOrderID);
        }
        Console.WriteLine("");
    }
}
```

```
Using context As New AdventureWorksEntities

    Dim query = context.SalesOrderHeaders _
        .GroupBy(Function(order) order.CustomerID)

    ' Iterate over each IGrouping
    For Each group In query

        Console.WriteLine("Customer ID: {0}", group.Key)
        Console.WriteLine("Order Count: {0}", group.Count)

        For Each sale In group
            Console.WriteLine("   Sale ID: {0}", sale.SalesOrderID)
        Next

        Console.WriteLine("")

    Next

End Using
```

See also

- Queries in LINQ to Entities

Method-Based Query Syntax Examples: Navigating Relationships

11/8/2022 • 4 minutes to read • Edit Online

Navigation properties in the Entity Framework are shortcut properties used to locate the entities at the ends of an association. Navigation properties allow a user to navigate from one entity to another, or from one entity to related entities through an association set. This topic provides examples in method-based query syntax of how to navigate relationships through navigation properties in LINQ to Entities queries.

The AdventureWorks Sales Model used in these examples is built from the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

```
Option Explicit On
Option Strict On
Imports System.Data.Objects
Imports System.Globalization
```

Example 1

The following example in method-based query syntax uses the SelectMany method to get all the orders of the contacts whose last name is "Zhou". The `Contact.SalesOrderHeader` navigation property is used to get the collection of `SalesOrderHeader` objects for each contact.

```
string lastName = "Zhou";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<SalesOrderHeader> ordersQuery = context.Contacts
        .Where(c => c.LastName == lastName)
        .SelectMany(c => c.SalesOrderHeaders);

    foreach (var order in ordersQuery)
    {
        Console.WriteLine("Order ID: {0}, Order date: {1}, Total Due: {2}",
            order.SalesOrderID, order.OrderDate, order.TotalDue);
    }
}
```

```
Dim lastName = "Zhou"
Using context As New AdventureWorksEntities
    Dim ordersQuery = context.Contacts _
    .Where(Function(c) c.LastName = lastName) _
    .SelectMany(Function(o) o.SalesOrderHeaders)

    For Each order In ordersQuery
        Console.WriteLine("Order ID: {0}, Order date: {1}, Total Due: {2}", _
                order.SalesOrderID, order.OrderDate, order.TotalDue)
    Next
End Using
```

Example 2

The following example in method-based query syntax uses the Select method to get all the contact IDs and the sum of the total due for each contact whose last name is "Zhou". The `Contact.SalesOrderHeader` navigation property is used to get the collection of `SalesOrderHeader` objects for each contact. The `Sum` method uses the `Contact.SalesOrderHeader` navigation property to sum the total due of all the orders for each contact.

```
string lastName = "Zhou";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var ordersQuery = context.Contacts
        .Where(c => c.LastName == lastName)
        .Select(c => new
        {
            ContactID = c.ContactID,
            Total = c.SalesOrderHeaders.Sum(o => o.TotalDue)
        });

    foreach (var contact in ordersQuery)
    {
        Console.WriteLine("Contact ID: {0} Orders total: {1}", contact.ContactID, contact.Total);
    }
}
```

```
Dim lastName = "Zhou"
Using context As New AdventureWorksEntities
    Dim ordersQuery = context.Contacts _
    .Where(Function(c) c.LastName = lastName) _
    .Select(Function(c) New With _
                {.ContactID = c.ContactID, _
                .Total = c.SalesOrderHeaders.Sum(Function(o) o.TotalDue)})

    For Each order In ordersQuery
        Console.WriteLine("Contact ID: {0} Orders total: {1}", order.ContactID, order.Total)
    Next
End Using
```

Example 3

The following example in method-based query syntax gets all the orders of the contacts whose last name is "Zhou". The `Contact.SalesOrderHeader` navigation property is used to get the collection of `SalesOrderHeader` objects for each contact. The contact's name and orders are returned in an anonymous type.

```
string lastName = "Zhou";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var ordersQuery = context.Contacts
        .Where(c => c.LastName == lastName)
        .Select(c => new { LastName = c.LastName, Orders = c.SalesOrderHeaders });

    foreach (var order in ordersQuery)
    {
        Console.WriteLine("Name: {0}", order.LastName);
        foreach (SalesOrderHeader orderInfo in order.Orders)
        {
            Console.WriteLine("Order ID: {0}, Order date: {1}, Total Due: {2}",
                orderInfo.SalesOrderID, orderInfo.OrderDate, orderInfo.TotalDue);
        }
        Console.WriteLine("");
    }
}
```

```
Dim lastName = "Zhou"
Using context As New AdventureWorksEntities
    Dim ordersQuery = context.Contacts _
    .Where(Function(c) c.LastName = lastName) _
    .Select(Function(o) New With _
                {.LastName = o.LastName, _
                 .Orders = o.SalesOrderHeaders})

    For Each order In ordersQuery
        Console.WriteLine("Name: {0}", order.LastName)
        For Each orderInfo In order.Orders

            Console.WriteLine("Order ID: {0}, Order date: {1}, Total Due: {2}", _
                    orderInfo.SalesOrderID, orderInfo.OrderDate, orderInfo.TotalDue)
        Next

        Console.WriteLine("")
    Next
End Using
```

Example 4

The following example uses the `SalesOrderHeader.Address` and `SalesOrderHeader.Contact` navigation properties to get the collection of `Address` and `Contact` objects associated with each order. The last name of the contact, the street address, the sales order number, and the total due for each order to the city of Seattle are returned in an anonymous type.

```
string city = "Seattle";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var ordersQuery = context.SalesOrderHeaders
        .Where(o => o.Address.City == city)
        .Select(o => new
        {
            ContactLastName = o.Contact.LastName,
            ContactFirstName = o.Contact.FirstName,
            StreetAddress = o.Address.AddressLine1,
            OrderNumber = o.SalesOrderNumber,
            TotalDue = o.TotalDue
        });

    foreach (var orderInfo in ordersQuery)
    {
        Console.WriteLine("Name: {0}, {1}", orderInfo.ContactLastName, orderInfo.ContactFirstName);
        Console.WriteLine("Street address: {0}", orderInfo.StreetAddress);
        Console.WriteLine("Order number: {0}", orderInfo.OrderNumber);
        Console.WriteLine("Total Due: {0}", orderInfo.TotalDue);
        Console.WriteLine("");
    }
}
```

```
Dim city = "Seattle"
Using context As New AdventureWorksEntities
    Dim ordersQuery = context.SalesOrderHeaders _
              .Where(Function(o) o.Address.City = city) _
              .Select(Function(o) New With { _
                            .ContactLastName = o.Contact.LastName, _
                            .ContactFirstName = o.Contact.FirstName, _
                            .StreetAddress = o.Address.AddressLine1, _
                            .OrderNumber = o.SalesOrderNumber, _
                            .TotalDue = o.TotalDue _
              })

    For Each orderInfo In ordersQuery
        Console.WriteLine("Name: {0}, {1}", orderInfo.ContactLastName, orderInfo.ContactFirstName)
        Console.WriteLine("Street address: {0}", orderInfo.StreetAddress)
        Console.WriteLine("Order number: {0}", orderInfo.OrderNumber)
        Console.WriteLine("Total Due: {0}", orderInfo.TotalDue)
        Console.WriteLine("")
    Next

End Using
```

Example 5

The following example uses the `Where` method to find orders that were made after December 1, 2003, and then uses the `order.SalesOrderDetail` navigation property to get the details for each order.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<SalesOrderHeader> query =
        from order in context.SalesOrderHeaders
        where order.OrderDate >= new DateTime(2003, 12, 1)
        select order;

    Console.WriteLine("Orders that were made after December 1, 2003:");
    foreach (SalesOrderHeader order in query)
    {
        Console.WriteLine("OrderID {0} Order date: {1:d} ",
            order.SalesOrderID, order.OrderDate);
        foreach (SalesOrderDetail orderDetail in order.SalesOrderDetails)
        {
            Console.WriteLine("  Product ID: {0} Unit Price {1}",
                orderDetail.ProductID, orderDetail.UnitPrice);
        }
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From order In orders _
        Where order.OrderDate >= New DateTime(2003, 12, 1) _
        Select order

    Console.WriteLine("Orders that were made after December 1, 2003:")
    For Each order In query
        Console.WriteLine("OrderID {0} Order date: {1:d} ", _
                order.SalesOrderID, order.OrderDate)
        For Each orderDetail In order.SalesOrderDetails
            Console.WriteLine("  Product ID: {0} Unit Price {1}", _
                orderDetail.ProductID, orderDetail.UnitPrice)
        Next
    Next
End Using
```

See also

- Relationships, navigation properties and foreign keys
- Queries in LINQ to Entities

Query Expression Syntax Examples: Projection

11/8/2022 • 5 minutes to read • Edit Online

The examples in this topic demonstrate how to use the `Select` method and the `From … From …` keywords to query the AdventureWorks Sales Model using query expression syntax. `From … From …` is the query based equivalent of the `SelectMany` method. The AdventureWorks Sales model used in these examples is built from the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

```
Option Explicit On
Option Strict On
Imports System.Data.Objects
Imports System.Globalization
```

Select

Example

The following example uses the Select method to return all the rows from the `Product` table and display the product names.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Product> productsQuery = from product in context.Products
                                        select product;

    Console.WriteLine("Product Names:");
    foreach (var prod in productsQuery)
    {
        Console.WriteLine(prod.Name);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim products As ObjectSet(Of Product) = context.Products

    Dim productsQuery = _
        From product In products _
        Select product

    Console.WriteLine("Product Names:")
    For Each product In productsQuery
        Console.WriteLine(product.Name)
    Next
End Using
```

Example

The following example uses Select to return a sequence of only product names.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<string> productNames =
        from p in context.Products
        select p.Name;

    Console.WriteLine("Product Names:");
    foreach (String productName in productNames)
    {
        Console.WriteLine(productName);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim products As ObjectSet(Of Product) = context.Products

    Dim productNames = _
        From p In products _
        Select p.Name

    Console.WriteLine("Product Names:")
    For Each productName In productNames
        Console.WriteLine(productName)
    Next
End Using
```

Example

The following example uses the Select method to project the `Product.Name` and `Product.ProductID` properties into a sequence of anonymous types.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query =
        from product in context.Products
        select new
        {
            ProductId = product.ProductID,
            ProductName = product.Name
        };

    Console.WriteLine("Product Info:");
    foreach (var productInfo in query)
    {
        Console.WriteLine("Product Id: {0} Product name: {1} ",
            productInfo.ProductId, productInfo.ProductName);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim products As ObjectSet(Of Product) = context.Products

    Dim query = _
        From product In products _
        Select New With _
        { _
            .ProductId = product.ProductID, _
            .ProductName = product.Name _
        }

    Console.WriteLine("Product Info:")
    For Each productInfo In query
        Console.WriteLine("Product Id: {0} Product name: {1} ", _
                productInfo.ProductId, productInfo.ProductName)
    Next
End Using
```

From … From … (SelectMany)

Example

The following example uses `From … From …` (the equivalent of the SelectMany method) to select all orders where `TotalDue` is less than 500.00.

```
decimal totalDue = 500.00M;
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from contact in contacts
        from order in orders
        where contact.ContactID == order.Contact.ContactID
            && order.TotalDue < totalDue
        select new
        {
            ContactID = contact.ContactID,
            LastName = contact.LastName,
            FirstName = contact.FirstName,
            OrderID = order.SalesOrderID,
            Total = order.TotalDue
        };

    foreach (var smallOrder in query)
    {
        Console.WriteLine("Contact ID: {0} Name: {1}, {2} Order ID: {3} Total Due: ${4} ",
            smallOrder.ContactID, smallOrder.LastName, smallOrder.FirstName,
            smallOrder.OrderID, smallOrder.Total);
    }
}
```

```
Dim totalDue = 500D
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From contact In contacts _
        From order In orders _
        Where contact.ContactID = order.Contact.ContactID _
                And order.TotalDue < totalDue _
        Select New With _
        { _
            .ContactID = contact.ContactID, _
            .LastName = contact.LastName, _
            .FirstName = contact.FirstName, _
            .OrderID = order.SalesOrderID, _
            .Total = order.TotalDue _
        }

    For Each smallOrder In query
        Console.WriteLine("Contact ID: {0} Name: {1}, {2} Order ID: {3} Total Due: ${4} ", _
            smallOrder.ContactID, smallOrder.LastName, smallOrder.FirstName, _
            smallOrder.OrderID, smallOrder.Total)
    Next
End Using
```

Example

The following example uses `From … From …` (the equivalent of the SelectMany method) to select all orders where the order was made on October 1, 2002 or later.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from contact in contacts
        from order in orders
        where contact.ContactID == order.Contact.ContactID
            && order.OrderDate >= new DateTime(2002, 10, 1)
        select new
        {
            ContactID = contact.ContactID,
            LastName = contact.LastName,
            FirstName = contact.FirstName,
            OrderID = order.SalesOrderID,
            OrderDate = order.OrderDate
        };

    foreach (var order in query)
    {
        Console.WriteLine("Contact ID: {0} Name: {1}, {2} Order ID: {3} Order date: {4:d} ",
            order.ContactID, order.LastName, order.FirstName,
            order.OrderID, order.OrderDate);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From contact In contacts _
        From order In orders _
        Where contact.ContactID = order.Contact.ContactID _
                And order.OrderDate >= New DateTime(2002, 10, 1) _
        Select New With _
        { _
            .ContactID = contact.ContactID, _
            .LastName = contact.LastName, _
            .FirstName = contact.FirstName, _
            .OrderID = order.SalesOrderID, _
            .OrderDate = order.OrderDate _
        }

    For Each order In query
        Console.WriteLine("Contact ID: {0} Name: {1}, {2} Order ID: {3} Order date: {4:d} ", _
            order.ContactID, order.LastName, order.FirstName, _
            order.OrderID, order.OrderDate)
    Next
End Using
```

Example

The following example uses a `From … From …` (the equivalent of the SelectMany method) to select all orders where the order total is greater than 10000.00 and uses `From` assignment to avoid requesting the total twice.

```
decimal totalDue = 10000.0M;
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from contact in contacts
        from order in orders
        let total = order.TotalDue
        where contact.ContactID == order.Contact.ContactID
            && total >= totalDue
        select new
        {
            ContactID = contact.ContactID,
            LastName = contact.LastName,
            OrderID = order.SalesOrderID,
            total
        };

    foreach (var order in query)
    {
        Console.WriteLine("Contact ID: {0} Last name: {1} Order ID: {2} Total: {3}",
            order.ContactID, order.LastName, order.OrderID, order.total);
    }
}
```

```
Dim totalDue = 10000D
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From contact In contacts _
        From order In orders _
        Let total = order.TotalDue _
        Where contact.ContactID = order.Contact.ContactID _
                And total >= totalDue _
        Select New With _
        { _
            .ContactID = contact.ContactID, _
            .LastName = contact.LastName, _
            .OrderID = order.SalesOrderID, _
            total _
        }

    For Each order In query
        Console.WriteLine("Contact ID: {0} Last name: {1} Order ID: {2} Total: {3}", _
                order.ContactID, order.LastName, order.OrderID, order.total)
    Next
End Using
```

See also

- Queries in LINQ to Entities

Query Expression Syntax Examples: Filtering

11/8/2022 • 4 minutes to read • Edit Online

The examples in this topic demonstrate how to use the `Where` and `Where…Contains` methods to query the AdventureWorks Sales Model using query expression syntax. Note, Where… `Contains` cannot be used as a part of a compiled query.

The AdventureWorks Sales model used in these examples is built from the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

```
Option Explicit On
Option Strict On
Imports System.Data.Objects
Imports System.Globalization
```

Where

Example

The following example returns all online orders.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var onlineOrders =
        from order in context.SalesOrderHeaders
        where order.OnlineOrderFlag == true
        select new
        {
            SalesOrderID = order.SalesOrderID,
            OrderDate = order.OrderDate,
            SalesOrderNumber = order.SalesOrderNumber
        };

    foreach (var onlineOrder in onlineOrders)
    {
        Console.WriteLine("Order ID: {0} Order date: {1:d} Order number: {2}",
            onlineOrder.SalesOrderID,
            onlineOrder.OrderDate,
            onlineOrder.SalesOrderNumber);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim onlineOrders = _
        From order In orders _
        Where order.OnlineOrderFlag = True _
        Select New With { _
           .SalesOrderID = order.SalesOrderID, _
           .OrderDate = order.OrderDate, _
           .SalesOrderNumber = order.SalesOrderNumber _
        }

    For Each onlineOrder In onlineOrders
        Console.WriteLine("Order ID: {0} Order date: {1:d} Order number: {2}", _
                onlineOrder.SalesOrderID, _
                onlineOrder.OrderDate, _
                onlineOrder.SalesOrderNumber)
    Next
End Using
```

Example

The following example returns the orders where the order quantity is greater than 2 and less than 6.

```
int orderQtyMin = 2;
int orderQtyMax = 6;
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query =
        from order in context.SalesOrderDetails
        where order.OrderQty > orderQtyMin && order.OrderQty < orderQtyMax
        select new
        {
            SalesOrderID = order.SalesOrderID,
            OrderQty = order.OrderQty
        };

    foreach (var order in query)
    {
        Console.WriteLine("Order ID: {0} Order quantity: {1}",
            order.SalesOrderID, order.OrderQty);
    }
}
```

```
Dim orderQtyMin = 2
Dim orderQtyMax = 6
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderDetail) = context.SalesOrderDetails

    Dim query = _
        From order In orders _
        Where order.OrderQty > orderQtyMin And order.OrderQty < orderQtyMax _
        Select New With { _
            .SalesOrderID = order.SalesOrderID, _
            .OrderQty = order.OrderQty _
        }

    For Each order In query
        Console.WriteLine("Order ID: {0} Order quantity: {1}", _
                order.SalesOrderID, order.OrderQty)
    Next
End Using
```

Example

The following example returns all red colored products.

```
String color = "Red";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query =
        from product in context.Products
        where product.Color == color
        select new
        {
            Name = product.Name,
            ProductNumber = product.ProductNumber,
            ListPrice = product.ListPrice
        };

    foreach (var product in query)
    {
        Console.WriteLine("Name: {0}", product.Name);
        Console.WriteLine("Product number: {0}", product.ProductNumber);
        Console.WriteLine("List price: ${0}", product.ListPrice);
        Console.WriteLine("");
    }
}
```

```
Dim color = "Red"
Using context As New AdventureWorksEntities
    Dim products As ObjectSet(Of Product) = context.Products

    Dim query = _
        From product In products _
        Where product.Color = color _
        Select New With { _
            .Name = product.Name, _
            .ProductNumber = product.ProductNumber, _
            .ListPrice = product.ListPrice _
        }

    For Each product In query
        Console.WriteLine("Name: {0}", product.Name)
        Console.WriteLine("Product number: {0}", product.ProductNumber)
        Console.WriteLine("List price: ${0}", product.ListPrice)
        Console.WriteLine("")
    Next
End Using
```

Example

The following example uses the `Where` method to find orders that were made after December 1, 2003, and then uses the `order.SalesOrderDetail` navigation property to get the details for each order.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<SalesOrderHeader> query =
        from order in context.SalesOrderHeaders
        where order.OrderDate >= new DateTime(2003, 12, 1)
        select order;

    Console.WriteLine("Orders that were made after December 1, 2003:");
    foreach (SalesOrderHeader order in query)
    {
        Console.WriteLine("OrderID {0} Order date: {1:d} ",
            order.SalesOrderID, order.OrderDate);
        foreach (SalesOrderDetail orderDetail in order.SalesOrderDetails)
        {
            Console.WriteLine("  Product ID: {0} Unit Price {1}",
                orderDetail.ProductID, orderDetail.UnitPrice);
        }
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From order In orders _
        Where order.OrderDate >= New DateTime(2003, 12, 1) _
        Select order

    Console.WriteLine("Orders that were made after December 1, 2003:")
    For Each order In query
        Console.WriteLine("OrderID {0} Order date: {1:d} ", _
                order.SalesOrderID, order.OrderDate)
        For Each orderDetail In order.SalesOrderDetails
            Console.WriteLine("  Product ID: {0} Unit Price {1}", _
                orderDetail.ProductID, orderDetail.UnitPrice)
        Next
    Next
End Using
```

Where...Contains

Example

The following example uses an array as part of a `Where...Contains` clause to find all products that have a `ProductModelID` that matches a value in the array.

```
using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
{
    int?[] productModelIds = {19, 26, 118};
    var products = from p in AWEntities.Products
                   where productModelIds.Contains(p.ProductModelID)
                   select p;
    foreach (var product in products)
    {
        Console.WriteLine("{0}: {1}", product.ProductModelID, product.ProductID);
    }
}
```

```
Using AWEntities As New AdventureWorksEntities()
    Dim productModelIds As System.Nullable(Of Integer)() = {19, 26, 118}
    Dim products = From p In AWEntities.Products _
                   Where productModelIds.Contains(p.ProductModelID) _
                   Select p
    For Each product In products
        Console.WriteLine("{0}: {1}", product.ProductModelID, product.ProductID)
    Next
End Using
```

> **NOTE**
>
> As part of the predicate in a `Where…Contains` clause, you can use an Array, a List<T>, or a collection of any type that implements the IEnumerable<T> interface. You can also declare and initialize a collection within a LINQ to Entities query. See the next example for more information.

Example

The following example declares and initializes arrays in a `Where…Contains` clause to find all products that have a `ProductModelID` or `Size` that match values in the arrays.

```
using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
{
    var products = from p in AWEntities.Products
                   where (new int?[] { 19, 26, 18 }).Contains(p.ProductModelID) ||
                         (new string[] { "L", "XL" }).Contains(p.Size)
                   select p;
    foreach (var product in products)
    {
        Console.WriteLine("{0}: {1}, {2}", product.ProductID,
                                           product.ProductModelID,
                                           product.Size);
    }
}
```

```
Using AWEntities As New AdventureWorksEntities()
    Dim products = From p In AWEntities.Products _
                   Where (New System.Nullable(Of Integer)() {19, 26, 18}).Contains(p.ProductModelID) _
                   OrElse (New String() {"L", "XL"}).Contains(p.Size) _
                   Select p
    For Each product In products
        Console.WriteLine("{0}: {1}, {2}", product.ProductID, _
                                           product.ProductModelID, _
                                           product.Size)
    Next
End Using
```

See also

- Queries in LINQ to Entities

Query Expression Syntax Examples: Ordering

11/8/2022 • 3 minutes to read • Edit Online

The examples in this topic demonstrate how to use the `OrderBy` and `OrderByDescending` methods to query the AdventureWorks Sales Model using query expression syntax. The AdventureWorks Sales Model used in these examples is built from the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

```
Option Explicit On
Option Strict On
Imports System.Data.Objects
Imports System.Globalization
```

OrderBy

Example

The following example uses OrderBy to return a list of contacts ordered by last name.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Contact> sortedNames =
        from n in context.Contacts
        orderby n.LastName
        select n;

    Console.WriteLine("The sorted list of last names:");
    foreach (Contact n in sortedNames)
    {
        Console.WriteLine(n.LastName);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts

    Dim sortedContacts = _
        From contact In contacts _
        Order By contact.LastName _
        Select contact

    Console.WriteLine("The sorted list of last names:")
    For Each n As Contact In sortedContacts
        Console.WriteLine(n.LastName)
    Next
End Using
```

Example

The following example uses OrderBy to sort a list of contacts by length of last name.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Contact> sortedNames =
        from n in context.Contacts
        orderby n.LastName.Length
        select n;

    Console.WriteLine("The sorted list of last names (by length):");
    foreach (Contact n in sortedNames)
    {
        Console.WriteLine(n.LastName);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts

    Dim sortedNames = _
        From n In contacts _
        Order By n.LastName.Length _
        Select n

    Console.WriteLine("The sorted list of last names (by length):")
    For Each n As Contact In sortedNames
        Console.WriteLine(n.LastName)
    Next
End Using
```

OrderByDescending

Example

The following example uses `orderby… descending` (`Order By … Descending` in Visual Basic), which is equivalent to the OrderByDescending method, to sort the price list from highest to lowest.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Decimal> sortedPrices =
        from p in context.Products
        orderby p.ListPrice descending
        select p.ListPrice;

    Console.WriteLine("The list price from highest to lowest:");
    foreach (Decimal price in sortedPrices)
    {
        Console.WriteLine(price);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim products As ObjectSet(Of Product) = context.Products

    Dim sortedPrices = _
        From product In products _
        Order By product.ListPrice Descending _
        Select product.ListPrice

    Console.WriteLine("The list price from highest to lowest:")
    For Each price As Decimal In sortedPrices
        Console.WriteLine(price)
    Next
End Using
```

ThenBy

Example

The following example uses OrderBy and ThenBy to return a list of contacts ordered by last name and then by first name.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Contact> sortedContacts =
        from contact in context.Contacts
        orderby contact.LastName, contact.FirstName
        select contact;

    Console.WriteLine("The list of contacts sorted by last name then by first name:");
    foreach (Contact sortedContact in sortedContacts)
    {
        Console.WriteLine(sortedContact.LastName + ", " + sortedContact.FirstName);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts

    Dim sortedContacts = _
        From contact In contacts _
        Order By contact.LastName, contact.FirstName _
        Select contact

    Console.WriteLine("The list of contacts sorted by last name then by first name:")
    For Each sortedContact As Contact In sortedContacts
        Console.WriteLine(sortedContact.LastName + ", " + sortedContact.FirstName)
    Next
End Using
```

ThenByDescending

Example

The following example uses `OrderBy… Descending`, which is equivalent to the ThenByDescending method, to sort a list of products, first by name and then by list price from highest to lowest.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Product> query =
        from product in context.Products
        orderby product.Name, product.ListPrice descending
        select product;

    foreach (Product product in query)
    {
        Console.WriteLine("Product ID: {0} Product Name: {1} List Price {2}",
            product.ProductID,
            product.Name,
            product.ListPrice);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim products As ObjectSet(Of Product) = context.Products

    Dim query As IQueryable(Of Product) = _
        From product In products _
        Order By product.Name, product.ListPrice Descending _
        Select product

    For Each prod As Product In query
        Console.WriteLine("Product ID: {0} Product Name: {1} List Price {2}", _
            prod.ProductID, _
            prod.Name, _
            prod.ListPrice)
    Next
End Using
```

See also

- Queries in LINQ to Entities

Query Expression Syntax Examples: Aggregate Operators

11/8/2022 • 8 minutes to read • Edit Online

The examples in this topic demonstrate how to use the Average, Count, Max, Min, and Sum methods to query the AdventureWorks Sales Model using query expression syntax. The AdventureWorks Sales Model used in these examples is built from the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

```
Option Explicit On
Option Strict On
Imports System.Data.Objects
Imports System.Globalization
```

Average

Example

The following example uses the Average method to find the average list price of the products of each style.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Product> products = context.Products;

    var query = from product in products
                group product by product.Style into g
                select new
                {
                    Style = g.Key,
                    AverageListPrice =
                        g.Average(product => product.ListPrice)
                };

    foreach (var product in query)
    {
        Console.WriteLine("Product style: {0} Average list price: {1}",
            product.Style, product.AverageListPrice);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim products As ObjectSet(Of Product) = context.Products

    Dim query = _
        From prod In products _
        Let styl = prod.Style _
        Group prod By styl Into g = Group _
        Select New With _
        { _
            .Style = styl, _
            .AverageListPrice = g.Average(Function(p) p.ListPrice) _
        }

    For Each prod In query
        Console.WriteLine("Product style: {0} Average list price: {1}", _
            prod.Style, prod.AverageListPrice)
    Next
End Using
```

Example

The following example uses Average to get the average total due for each contact ID.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from order in orders
        group order by order.Contact.ContactID into g
        select new
        {
            Category = g.Key,
            averageTotalDue = g.Average(order => order.TotalDue)
        };

    foreach (var order in query)
    {
        Console.WriteLine("ContactID = {0} \t Average TotalDue = {1}",
            order.Category, order.averageTotalDue);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From ord In orders _
        Let contID = ord.Contact.ContactID _
        Group ord By contID Into g = Group _
        Select New With _
        { _
            .Category = contID, _
            .averageTotalDue = _
                g.Average(Function(ord) ord.TotalDue) _
         }

    For Each ord In query
        Console.WriteLine("ContactID = {0} " & vbTab & _
            " Average TotalDue = {1}", _
            ord.Category, ord.averageTotalDue)
    Next
End Using
```

Example

The following example uses Average to get the orders with the average total due for each contact.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from order in orders
        group order by order.Contact.ContactID into g
        let averageTotalDue = g.Average(order => order.TotalDue)
        select new
        {
            Category = g.Key,
            CheapestProducts =
                g.Where(order => order.TotalDue == averageTotalDue)
        };

    foreach (var orderGroup in query)
    {
        Console.WriteLine("ContactID: {0}", orderGroup.Category);
        foreach (var order in orderGroup.CheapestProducts)
        {
            Console.WriteLine("Average total due for SalesOrderID {1} is: {0}",
                order.TotalDue, order.SalesOrderID);
        }
        Console.Write("\n");
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From ord In orders _
        Let contID = ord.Contact.ContactID _
        Group ord By contID Into g = Group _
        Let averageTotalDue = g.Average(Function(ord) ord.TotalDue) _
        Select New With _
        { _
            .Category = contID, _
            .CheapestProducts = _
                g.Where(Function(ord) ord.TotalDue = averageTotalDue) _
         }

    For Each orderGroup In query
        Console.WriteLine("ContactID: {0}", orderGroup.Category)
        For Each ord In orderGroup.CheapestProducts
            Console.WriteLine("Average total due for SalesOrderID {1} is: {0}", _
                ord.TotalDue, ord.SalesOrderID)
        Next
        Console.Write(vbNewLine)
    Next
End Using
```

Count

Example

The following example uses Count to return a list of contact IDs and how many orders each has.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;

    //Can't find field SalesOrderContact
    var query =
        from contact in contacts
        select new
        {
            CustomerID = contact.ContactID,
            OrderCount = contact.SalesOrderHeaders.Count()
        };

    foreach (var contact in query)
    {
        Console.WriteLine("CustomerID = {0} \t OrderCount = {1}",
            contact.CustomerID,
            contact.OrderCount);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts

    Dim query = _
        From cont In contacts _
        Select New With _
        { _
            .CustomerID = cont.ContactID, _
            .OrderCount = cont.SalesOrderHeaders.Count() _
         }

    For Each cont In query
        Console.WriteLine("CustomerID = {0}   OrderCount = {1}", _
               cont.CustomerID, cont.OrderCount)
    Next
End Using
```

Example

The following example groups products by color and uses Count to return the number of products in each color group.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Product> products = context.Products;

    var query =
        from product in products
        group product by product.Color into g
        select new { Color = g.Key, ProductCount = g.Count() };

    foreach (var product in query)
    {
        Console.WriteLine("Color = {0} \t ProductCount = {1}",
            product.Color,
            product.ProductCount);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim products As ObjectSet(Of Product) = context.Products

    Dim query = _
        From prod In products _
        Let pc = prod.Color _
        Group prod By pc Into g = Group _
        Select New With {.Color = pc, .ProductCount = g.Count()}

    For Each prod In query
        Console.WriteLine("Color = {0} " & vbTab & " ProductCount = {1}", _
            prod.Color, prod.ProductCount)
    Next
End Using
```

Max

Example

The following example uses the Max method to get the largest total due for each contact ID.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from order in orders
        group order by order.Contact.ContactID into g
        select new
        {
            Category = g.Key,
            maxTotalDue =
                g.Max(order => order.TotalDue)
        };

    foreach (var order in query)
    {
        Console.WriteLine("ContactID = {0} \t Maximum TotalDue = {1}",
            order.Category, order.maxTotalDue);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From ord In orders _
        Let contID = ord.Contact.ContactID _
        Group ord By contID Into g = Group _
        Select New With _
        { _
            .Category = contID, _
            .MaxTotalDue = _
                g.Max(Function(ord) ord.TotalDue) _
         }

    For Each ord In query
        Console.WriteLine("ContactID = {0} " & vbTab & _
            " Maximum TotalDue = {1}", _
            ord.Category, ord.MaxTotalDue)
    Next
End Using
```

Example

The following example uses the Max method to get the orders with the largest total due for each contact ID.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from order in orders
        group order by order.Contact.ContactID into g
        let maxTotalDue = g.Max(order => order.TotalDue)
        select new
        {
            Category = g.Key,
            CheapestProducts =
                g.Where(order => order.TotalDue == maxTotalDue)
        };

    foreach (var orderGroup in query)
    {
        Console.WriteLine("ContactID: {0}", orderGroup.Category);
        foreach (var order in orderGroup.CheapestProducts)
        {
            Console.WriteLine("MaxTotalDue {0} for SalesOrderID {1}: ",
                order.TotalDue,
                order.SalesOrderID);
        }
        Console.Write("\n");
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From ord In orders _
        Let contID = ord.Contact.ContactID _
        Group ord By contID Into g = Group _
        Let maxTotalDue = g.Max(Function(ord) ord.TotalDue) _
        Select New With _
        { _
            .Category = contID, _
            .CheapestProducts = _
                g.Where(Function(ord) ord.TotalDue = maxTotalDue) _
         }

    For Each orderGroup In query
        Console.WriteLine("ContactID: {0}", orderGroup.Category)
        For Each ord In orderGroup.CheapestProducts
            Console.WriteLine("MaxTotalDue {0} for SalesOrderID {1}: ", _
                ord.TotalDue, ord.SalesOrderID)
        Next
        Console.Write(vbNewLine)
    Next
End Using
```

Min

Example

The following example uses the Min method to get the smallest total due for each contact ID.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from order in orders
        group order by order.Contact.ContactID into g
        select new
        {
            Category = g.Key,
            smallestTotalDue =
                g.Min(order => order.TotalDue)
        };

    foreach (var order in query)
    {
        Console.WriteLine("ContactID = {0} \t Minimum TotalDue = {1}",
            order.Category, order.smallestTotalDue);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From ord In orders _
        Let contID = ord.Contact.ContactID _
        Group ord By contID Into g = Group _
        Select New With _
        { _
            .Category = contID, _
            .smallestTotalDue = _
                g.Min(Function(o) o.TotalDue) _
         }

    For Each ord In query
        Console.WriteLine("ContactID = {0} " & vbTab & _
            " Minimum TotalDue = {1}", ord.Category, ord.smallestTotalDue)
    Next
End Using
```

Example

The following example uses the Min method to get the orders with the smallest total due for each contact.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from order in orders
        group order by order.Contact.ContactID into g
        let minTotalDue = g.Min(order => order.TotalDue)
        select new
        {
            Category = g.Key,
            smallestTotalDue =
                g.Where(order => order.TotalDue == minTotalDue)
        };

    foreach (var orderGroup in query)
    {
        Console.WriteLine("ContactID: {0}", orderGroup.Category);
        foreach (var order in orderGroup.smallestTotalDue)
        {
            Console.WriteLine("Mininum TotalDue {0} for SalesOrderID {1}: ",
                order.TotalDue,
                order.SalesOrderID);
        }
        Console.Write("\n");
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From ord In orders _
        Let contID = ord.Contact.ContactID _
        Group ord By contID Into g = Group _
        Let minTotalDue = g.Min(Function(o) o.TotalDue) _
        Select New With _
        { _
            .Category = contID, _
            .smallestTotalDue = _
                g.Where(Function(o) o.TotalDue = minTotalDue) _
         }

    For Each orderGroup In query
        Console.WriteLine("ContactID: {0}", orderGroup.Category)
        For Each ord In orderGroup.smallestTotalDue
            Console.WriteLine("Mininum TotalDue {0} for SalesOrderID {1}: ", _
                ord.TotalDue, ord.SalesOrderID)
        Next
        Console.Write(vbNewLine)
    Next
End Using
```

Sum

Example

The following example uses the Sum method to get the total due for each contact ID.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from order in orders
        group order by order.Contact.ContactID into g
        select new
        {
            Category = g.Key,
            TotalDue = g.Sum(order => order.TotalDue)
        };

    foreach (var order in query)
    {
        Console.WriteLine("ContactID = {0} \t TotalDue sum = {1}",
            order.Category, order.TotalDue);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From ord In orders _
        Let contID = ord.Contact.ContactID _
        Group ord By contID Into g = Group _
        Select New With _
        { _
            .Category = contID, _
            .TotalDue = g.Sum(Function(o) o.TotalDue) _
        }

    For Each ord In query
        Console.WriteLine("ContactID = {0} " & vbTab & _
            " TotalDue sum = {1}", ord.Category, ord.TotalDue)
    Next
End Using
```

See also

- Queries in LINQ to Entities

Query Expression Syntax Examples: Partitioning

11/8/2022 • 2 minutes to read • Edit Online

The examples in this topic demonstrate how to use the Skip and Take methods to query the AdventureWorks Sales Model using query expression syntax. The AdventureWorks Sales Model used in these examples is built from the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

```
Option Explicit On
Option Strict On
Imports System.Data.Objects
Imports System.Globalization
```

Skip

Example

The following example uses the Skip method to get all but the first two addresses in Seattle.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Address> addresses = context.Addresses;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    //LINQ to Entities only supports Skip on ordered collections.
    var query = (
        from address in addresses
        from order in orders
        where address.AddressID == order.Address.AddressID
             && address.City == "Seattle"
        orderby order.SalesOrderID
        select new
        {
            City = address.City,
            OrderID = order.SalesOrderID,
            OrderDate = order.OrderDate
        }).Skip(2);

    Console.WriteLine("All but first 2 orders in Seattle:");
    foreach (var order in query)
    {
        Console.WriteLine("City: {0} Order ID: {1} Total Due: {2:d}",
            order.City, order.OrderID, order.OrderDate);
    }
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders
    Dim addresses As ObjectSet(Of Address) = context.Addresses

    'LINQ to Entities only supports Skip on ordered collections.
    Dim query = ( _
        From address In addresses _
        From order In orders _
        Where address.AddressID = order.Address.AddressID _
                 And address.City = "Seattle" _
        Order By order.SalesOrderID _
        Select New With _
        { _
            .City = address.City, _
            .OrderID = order.SalesOrderID, _
            .OrderDate = order.OrderDate _
        }).Skip(2)

    Console.WriteLine("All but first 2 orders in Seattle:")
    For Each order In query
        Console.WriteLine("City: {0} Order ID: {1} Total Due: {2:d}", _
            order.City, order.OrderID, order.OrderDate)
    Next
End Using
```

Take

Example

The following example uses the Take method to get the first three addresses in Seattle.

```
String city = "Seattle";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Address> addresses = context.Addresses;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query = (
        from address in addresses
        from order in orders
        where address.AddressID == order.Address.AddressID
             && address.City == city
        select new
        {
            City = address.City,
            OrderID = order.SalesOrderID,
            OrderDate = order.OrderDate
        }).Take(3);
    Console.WriteLine("First 3 orders in Seattle:");
    foreach (var order in query)
    {
        Console.WriteLine("City: {0} Order ID: {1} Total Due: {2:d}",
            order.City, order.OrderID, order.OrderDate);
    }
}
```

```
Dim city = "Seattle"
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders
    Dim addresses As ObjectSet(Of Address) = context.Addresses

    Dim query = ( _
        From address In addresses _
        From order In orders _
        Where address.AddressID = order.Address.AddressID _
                 And address.City = city _
        Select New With _
        { _
            .City = address.City, _
            .OrderID = order.SalesOrderID, _
            .OrderDate = order.OrderDate _
        }).Take(3)

    Console.WriteLine("First 3 orders in Seattle:")
    For Each order In query
        Console.WriteLine("City: {0} Order ID: {1} Total Due: {2:d}", _
            order.City, order.OrderID, order.OrderDate)
    Next
End Using
```

See also

- Queries in LINQ to Entities

Query Expression Syntax Examples: Join Operators

11/8/2022 • 3 minutes to read • Edit Online

Joining is an important operation in queries that target data sources that have no navigable relationships to each other, such as relational database tables. A join of two data sources is the association of objects in one data source with objects that share a common attribute in the other data source. For more information, see Standard Query Operators Overview.

The examples in this topic demonstrate how to use the GroupJoin and Join methods to query the AdventureWorks Sales Model using query expression syntax. The AdventureWorks Sales Model used in these examples is built from the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

```
Option Explicit On
Option Strict On
Imports System.Data.Objects
Imports System.Globalization
```

GroupJoin

Example

The following example performs a GroupJoin over the SalesOrderHeader and SalesOrderDetail tables to find the number of orders per customer. A group join is the equivalent of a left outer join, which returns each element of the first (left) data source, even if no correlated elements are in the other data source.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;
    ObjectSet<SalesOrderDetail> details = context.SalesOrderDetails;

    var query =
        from order in orders
        join detail in details
        on order.SalesOrderID
        equals detail.SalesOrderID into orderGroup
        select new
        {
            CustomerID = order.SalesOrderID,
            OrderCount = orderGroup.Count()
        };

    foreach (var order in query)
    {
        Console.WriteLine("CustomerID: {0}  Orders Count: {1}",
            order.CustomerID,
            order.OrderCount);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders
    Dim details As ObjectSet(Of SalesOrderDetail) = context.SalesOrderDetails

    Dim query = _
        From order In orders _
        Group Join detail In details _
        On order.SalesOrderID _
        Equals detail.SalesOrderID Into orderGroup = Group _
        Select New With _
        { _
            .CustomerID = order.SalesOrderID, _
            .OrderCount = orderGroup.Count() _
        }

    For Each order In query
        Console.WriteLine("CustomerID: {0}  Orders Count: {1}", _
            order.CustomerID, order.OrderCount)
    Next
End Using
```

Example

The following example performs a GroupJoin over the Contact and SalesOrderHeader tables to find the number of orders per contact. The order count and IDs for each contact are displayed.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var query =
        from contact in contacts
        join order in orders
        on contact.ContactID
        equals order.Contact.ContactID into contactGroup
        select new
        {
            ContactID = contact.ContactID,
            OrderCount = contactGroup.Count(),
            Orders = contactGroup
        };

    foreach (var group in query)
    {
        Console.WriteLine("ContactID: {0}", group.ContactID);
        Console.WriteLine("Order count: {0}", group.OrderCount);
        foreach (var orderInfo in group.Orders)
        {
            Console.WriteLine("   Sale ID: {0}", orderInfo.SalesOrderID);
        }
        Console.WriteLine("");
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From contact In contacts _
        Group Join order In orders _
        On contact.ContactID _
        Equals order.Contact.ContactID Into contactGroup = Group _
        Select New With { _
            .ContactID = contact.ContactID, _
            .OrderCount = contactGroup.Count(), _
            .Orders = contactGroup.Select(Function(order) order)}

    For Each group In query
        Console.WriteLine("ContactID: {0}", group.ContactID)
        Console.WriteLine("Order count: {0}", group.OrderCount)

        For Each orderInfo In group.Orders
            Console.WriteLine("   Sale ID: {0}", orderInfo.SalesOrderID)
        Next

        Console.WriteLine("")
    Next
End Using
```

Join

Example

The following example performs a join over the SalesOrderHeader and SalesOrderDetail tables to get online orders from the month of August.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;
    ObjectSet<SalesOrderDetail> details = context.SalesOrderDetails;

    var query =
        from order in orders
        join detail in details
        on order.SalesOrderID equals detail.SalesOrderID
        where order.OnlineOrderFlag == true
        && order.OrderDate.Month == 8
        select new
        {
            SalesOrderID = order.SalesOrderID,
            SalesOrderDetailID = detail.SalesOrderDetailID,
            OrderDate = order.OrderDate,
            ProductID = detail.ProductID
        };

    foreach (var order in query)
    {
        Console.WriteLine("{0}\t{1}\t{2:d}\t{3}",
            order.SalesOrderID,
            order.SalesOrderDetailID,
            order.OrderDate,
            order.ProductID);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders
    Dim details As ObjectSet(Of SalesOrderDetail) = context.SalesOrderDetails

    Dim query = _
        From ord In orders _
        Join det In details _
        On ord.SalesOrderID Equals det.SalesOrderID _
        Where ord.OnlineOrderFlag = True _
                And ord.OrderDate.Month = 8 _
        Select New With _
        { _
            .SalesOrderID = ord.SalesOrderID, _
            .SalesOrderDetailID = det.SalesOrderDetailID, _
            .OrderDate = ord.OrderDate, _
            .ProductID = det.ProductID _
        }

    For Each ord In query
        Console.WriteLine("{0}" & vbTab & "{1}" & vbTab & "{2:d}" & vbTab & "{3}", _
            ord.SalesOrderID, _
            ord.SalesOrderDetailID, _
            ord.OrderDate, _
            ord.ProductID)
    Next
End Using
```

See also

- Queries in LINQ to Entities

Query Expression Syntax Examples: Element Operators

11/8/2022 • 2 minutes to read • Edit Online

The examples in this topic demonstrate how to use the First method to query the AdventureWorks Sales Model using the query expression syntax. The AdventureWorks Sales Model used in these examples is built from the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

```
Option Explicit On
Option Strict On
Imports System.Data.Objects
Imports System.Globalization
```

First

Example

The following example uses the First method to return the first contact whose first name is "Brooke".

```
string firstName = "Brooke";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;

    Contact query = (
        from contact in contacts
        where contact.FirstName == firstName
        select contact)
        .First();

    Console.WriteLine("ContactID: " + query.ContactID);
    Console.WriteLine("FirstName: " + query.FirstName);
    Console.WriteLine("LastName: " + query.LastName);
}
```

```
Dim firstName = "Brooke"
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts

    Dim query As Contact = ( _
        From cont In contacts _
        Where cont.FirstName = firstName _
        Select cont).First()

    Console.WriteLine("ContactID: " & query.ContactID)
    Console.WriteLine("FirstName: " & query.FirstName)
    Console.WriteLine("LastName: " & query.LastName)
End Using
```

See also

- Queries in LINQ to Entities

Query Expression Syntax Examples: Grouping

11/8/2022 • 2 minutes to read • Edit Online

The examples in this topic demonstrate how to use the `GroupBy` method to query the AdventureWorks Sales Model using query expression syntax. The AdventureWorks Sales model used in these examples is built from the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

```
Option Explicit On
Option Strict On
Imports System.Data.Objects
Imports System.Globalization
```

Example 1

The following example returns `Address` objects grouped by postal code. The results are projected into an anonymous type.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query =
        from address in context.Addresses
        group address by address.PostalCode into addressGroup
        select new { PostalCode = addressGroup.Key,
                     AddressLine = addressGroup };

    foreach (var addressGroup in query)
    {
        Console.WriteLine("Postal Code: {0}", addressGroup.PostalCode);
        foreach (var address in addressGroup.AddressLine)
        {
            Console.WriteLine("\t" + address.AddressLine1 +
                address.AddressLine2);
        }
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim addresses As ObjectSet(Of Address) = context.Addresses

    Dim query = _
        From adrs In addresses _
        Group adrs By adrs.PostalCode Into g = Group _
        Select New With {.PostalCode = PostalCode, .AddressLine = g}

    For Each addressGroup In query
        Console.WriteLine("Postal Code: {0}", addressGroup.PostalCode)
        For Each adrs In addressGroup.AddressLine
            Console.WriteLine(vbTab & adrs.AddressLine1 & _
                adrs.AddressLine2)
        Next
    Next
End Using
```

Example 2

The following example returns `Contact` objects grouped by the first letter of the contact's last name. The results are also sorted by the first letter of last name and projected into an anonymous type.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query = (
        from contact in context.Contacts
        group contact by contact.LastName.Substring(0, 1) into contactGroup
        select new { FirstLetter = contactGroup.Key, Names = contactGroup }).
            OrderBy(letter => letter.FirstLetter);

    foreach (var contact in query)
    {
        Console.WriteLine("Last names that start with the letter '{0}':",
            contact.FirstLetter);
        foreach (var name in contact.Names)
        {
            Console.WriteLine(name.LastName);
        }
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts

    Dim query = ( _
        From contact In contacts _
        Group By firstLetter = contact.LastName.Substring(0, 1) _
        Into contactGroup = Group _
        Select New With {.FirstLetter = firstLetter, .Names = contactGroup}) _
        .OrderBy(Function(letter) letter.FirstLetter)

    For Each n In query
        Console.WriteLine("Last names that start with the letter '{0}':", _
            n.FirstLetter)
        For Each name In n.Names
            Console.WriteLine(name.LastName)
        Next
    Next
End Using
```

Example 3

The following example returns `SalesOrderHeader` objects grouped by customer ID. The number of sales for each customer is also returned.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var query = from order in context.SalesOrderHeaders
                group order by order.CustomerID into idGroup
                select new {CustomerID = idGroup.Key,
                    OrderCount = idGroup.Count(),
                    Sales = idGroup};

    foreach (var orderGroup in query)
    {
        Console.WriteLine("Customer ID: {0}", orderGroup.CustomerID);
        Console.WriteLine("Order Count: {0}", orderGroup.OrderCount);

        foreach (SalesOrderHeader sale in orderGroup.Sales)
        {
            Console.WriteLine("   Sale ID: {0}", sale.SalesOrderID);
        }

        Console.WriteLine("");
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim salesOrders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = From order In salesOrders _
                Group order By order.CustomerID Into idGroup = Group, Count()

    For Each group In query
        Console.WriteLine("Customer ID: {0}", group.CustomerID)
        Console.WriteLine("Order Count: {0}", group.Count)

        For Each sale In group.idGroup
            Console.WriteLine("   Sale ID: {0}", sale.SalesOrderID)
        Next

        Console.WriteLine("")
    Next


End Using
```

See also

- Queries in LINQ to Entities

Query Expression Syntax Examples: Navigating Relationships

11/8/2022 • 4 minutes to read • Edit Online

Navigation properties in the Entity Framework are shortcut properties used to locate the entities at the ends of an association. Navigation properties allow a user to navigate from one entity to another, or from one entity to related entities through an association set. This topic provides examples in query expression syntax of how to navigate relationships through navigation properties in LINQ to Entities queries.

The AdventureWorks Sales Model used in these examples is built from the Contact, Address, Product, SalesOrderHeader, and SalesOrderDetail tables in the AdventureWorks sample database.

The examples in this topic use the following `using` / `Imports` statements:

```
using System;
using System.Data;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Objects;
using System.Globalization;
using System.Data.EntityClient;
using System.Data.SqlClient;
using System.Data.Common;
```

```
Option Explicit On
Option Strict On
Imports System.Data.Objects
Imports System.Globalization
```

Example 1

The following example uses the Select method to get all the contact IDs and the sum of the total due for each contact whose last name is "Zhou". The `Contact.SalesOrderHeader` navigation property is used to get the collection of `SalesOrderHeader` objects for each contact. The `Sum` method uses the `Contact.SalesOrderHeader` navigation property to sum the total due of all the orders for each contact.

```
string lastName = "Zhou";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;

    var ordersQuery = from contact in contacts
                      where contact.LastName == lastName
                      select new
                      {
                          ContactID = contact.ContactID,
                          Total = contact.SalesOrderHeaders.Sum(o => o.TotalDue)
                      };

    foreach (var contact in ordersQuery)
    {
        Console.WriteLine("Contact ID: {0} Orders total: {1}", contact.ContactID, contact.Total);
    }
}
```

```
Dim lastName = "Zhou"
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts

    Dim ordersQuery = From contact In contacts _
                      Where contact.LastName = lastName _
                      Select New With _
                              {.ContactID = contact.ContactID, _
                              .Total = contact.SalesOrderHeaders.Sum(Function(o) o.TotalDue)}

    For Each order In ordersQuery
        Console.WriteLine("Contact ID: {0} Orders total: {1}", order.ContactID, order.Total)
    Next
End Using
```

Example 2

The following example gets all the orders of the contacts whose last name is "Zhou". The `Contact.SalesOrderHeader` navigation property is used to get the collection of `SalesOrderHeader` objects for each contact. The contact's name and orders are returned in an anonymous type.

```
string lastName = "Zhou";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Contact> contacts = context.Contacts;

    var ordersQuery = from contact in contacts
                      where contact.LastName == lastName
                      select new { LastName = contact.LastName, Orders = contact.SalesOrderHeaders };

    foreach (var order in ordersQuery)
    {
        Console.WriteLine("Name: {0}", order.LastName);
        foreach (SalesOrderHeader orderInfo in order.Orders)
        {
            Console.WriteLine("Order ID: {0}, Order date: {1}, Total Due: {2}",
                orderInfo.SalesOrderID, orderInfo.OrderDate, orderInfo.TotalDue);
        }
        Console.WriteLine("");
    }
}
```

```
Dim lastName = "Zhou"
Using context As New AdventureWorksEntities
    Dim contacts As ObjectSet(Of Contact) = context.Contacts

    Dim ordersQuery = From contact In contacts _
                      Where contact.LastName = lastName _
                      Select New With _
                              {.LastName = contact.LastName, _
                               .Orders = contact.SalesOrderHeaders}

    For Each order In ordersQuery
        Console.WriteLine("Name: {0}", order.LastName)
        For Each orderInfo In order.Orders

            Console.WriteLine("Order ID: {0}, Order date: {1}, Total Due: {2}", _
                    orderInfo.SalesOrderID, orderInfo.OrderDate, orderInfo.TotalDue)
        Next

        Console.WriteLine("")
    Next
End Using
```

Example 3

The following example uses the `SalesOrderHeader.Address` and `SalesOrderHeader.Contact` navigation properties get the collection of `Address` and `Contact` objects associated with each order. The last name of the contact, the street address, the sales order number, and the total due for each order to the city of Seattle are returned in an anonymous type.

```
string city = "Seattle";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;

    var ordersQuery = from order in orders
                      where order.Address.City == city
                      select new
                      {
                          ContactLastName = order.Contact.LastName,
                          ContactFirstName = order.Contact.FirstName,
                          StreetAddress = order.Address.AddressLine1,
                          OrderNumber = order.SalesOrderNumber,
                          TotalDue = order.TotalDue
                      };

    foreach (var orderInfo in ordersQuery)
    {
        Console.WriteLine("Name: {0}, {1}", orderInfo.ContactLastName, orderInfo.ContactFirstName);
        Console.WriteLine("Street address: {0}", orderInfo.StreetAddress);
        Console.WriteLine("Order number: {0}", orderInfo.OrderNumber);
        Console.WriteLine("Total Due: {0}", orderInfo.TotalDue);
        Console.WriteLine("");
    }
}
```

```
Dim city = "Seattle"
Using context As New AdventureWorksEntities

    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim ordersQuery = From order In orders _
                      Where order.Address.City = city _
                      Select New With { _
                                   .ContactLastName = order.Contact.LastName, _
                                   .ContactFirstName = order.Contact.FirstName, _
                                   .StreetAddress = order.Address.AddressLine1, _
                                   .OrderNumber = order.SalesOrderNumber, _
                                   .TotalDue = order.TotalDue}

    For Each orderInfo In ordersQuery
        Console.WriteLine("Name: {0}, {1}", orderInfo.ContactLastName, orderInfo.ContactFirstName)
        Console.WriteLine("Street address: {0}", orderInfo.StreetAddress)
        Console.WriteLine("Order number: {0}", orderInfo.OrderNumber)
        Console.WriteLine("Total Due: {0}", orderInfo.TotalDue)
        Console.WriteLine("")
    Next

End Using
```

Example 4

The following example uses the `Where` method to find orders that were made after December 1, 2003, and then uses the `order.SalesOrderDetail` navigation property to get the details for each order.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<SalesOrderHeader> query =
        from order in context.SalesOrderHeaders
        where order.OrderDate >= new DateTime(2003, 12, 1)
        select order;

    Console.WriteLine("Orders that were made after December 1, 2003:");
    foreach (SalesOrderHeader order in query)
    {
        Console.WriteLine("OrderID {0} Order date: {1:d} ",
            order.SalesOrderID, order.OrderDate);
        foreach (SalesOrderDetail orderDetail in order.SalesOrderDetails)
        {
            Console.WriteLine("  Product ID: {0} Unit Price {1}",
                orderDetail.ProductID, orderDetail.UnitPrice);
        }
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders

    Dim query = _
        From order In orders _
        Where order.OrderDate >= New DateTime(2003, 12, 1) _
        Select order

    Console.WriteLine("Orders that were made after December 1, 2003:")
    For Each order In query
        Console.WriteLine("OrderID {0} Order date: {1:d} ", _
                order.SalesOrderID, order.OrderDate)
        For Each orderDetail In order.SalesOrderDetails
            Console.WriteLine("  Product ID: {0} Unit Price {1}", _
                orderDetail.ProductID, orderDetail.UnitPrice)
        Next
    Next
End Using
```

See also

- Queries in LINQ to Entities

Expressions in LINQ to Entities Queries

11/8/2022 • 2 minutes to read • Edit Online

An expression is a fragment of code that can be evaluated to a single value, object, method, or namespace. Expressions can contain a literal value, a method call, an operator and its operands, or a simple name. Simple names can be the name of a variable, type member, method parameter, namespace or type. Expressions can use operators that in turn use other expressions as parameters, or method calls whose parameters are in turn other method calls. Therefore, expressions can range from simple to very complex.

In LINQ to Entities queries, expressions can contain anything allowed by the types within the System.Linq.Expressions namespace, including lambda expressions. The expressions that can be used in LINQ to Entities queries are a superset of the expressions that can be used to query the Entity Framework.Expressions that are part of queries against the Entity Framework are limited to operations supported by `ObjectQuery<T>` and the underlying data source.

In the following example, the comparison in the `Where` clause is an expression:

```
Decimal totalDue = 200;
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<int> salesInfo =
        from s in context.SalesOrderHeaders
        where s.TotalDue >= totalDue
        select s.SalesOrderID;

    Console.WriteLine("Sales order info:");
    foreach (int orderNumber in salesInfo)
    {
        Console.WriteLine("Order number: " + orderNumber);
    }
}
```

```
Dim totalDue = 200
Using context As New AdventureWorksEntities()
    Dim salesInfo = _
        From s In context.SalesOrderHeaders _
        Where s.TotalDue >= totalDue _
        Select s.SalesOrderID

    Console.WriteLine("Sales order info:")
    For Each orderNumber As Integer In salesInfo
        Console.WriteLine("Order number: " & orderNumber)
    Next
End Using
```

NOTE

Specific language constructs, such as C# `unchecked`, have no meaning in LINQ to Entities.

In This Section

Constant Expressions

Comparison Expressions

Null Comparisons

Initialization Expressions

Relationships, navigation properties and foreign keys

See also

- ADO.NET Entity Framework

Constant Expressions

11/8/2022 • 2 minutes to read • Edit Online

A constant expression consists of a constant value. Constant values are directly converted to constant command tree expressions, without any translation on the client. This includes expressions that result in a constant value. Therefore, data source behavior should be expected for all expressions involving constants. This can result in behavior that differs from CLR behavior.

The following example shows a constant expression that is evaluated on the server.

```
Decimal totalDue = 200 + 3;
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<string> salesInfo =
        from s in context.SalesOrderHeaders
        where s.TotalDue >= totalDue
        select s.SalesOrderNumber;

    Console.WriteLine("Sales order numbers:");
    foreach (string orderNum in salesInfo)
    {
        Console.WriteLine(orderNum);
    }
}
```

```
Dim totalDue = 200 + 3
Using context As New AdventureWorksEntities()
    Dim salesInfo = _
        From s In context.SalesOrderHeaders _
        Where s.TotalDue >= totalDue _
        Select s.SalesOrderNumber

    Console.WriteLine("Sales order numbers:")
    For Each orderNum As String In salesInfo
        Console.WriteLine(orderNum)
    Next
End Using
```

LINQ to Entities does not support using a user class as a constant. However, a property reference on a user class is considered a constant, and will be converted to a command tree constant expression and executed on the data source.

See also

- Expressions in LINQ to Entities Queries

Comparison Expressions

11/8/2022 • 4 minutes to read • Edit Online

A comparison expression checks whether a constant value, property value, or method result is equal, not equal, greater than, or less than another value. If a particular comparison is not valid for LINQ to Entities, an exception will be thrown. All comparisons, both implicit and explicit, require that all components are comparable in the data source. Comparison expressions are frequently used in `Where` clauses for restricting the query results.

The following example in query expression syntax shows a query that returns results where the sales order number is equal to "SO43663":

```
string salesOrderNumber = "SO43663";
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<SalesOrderHeader> salesInfo =
        from s in context.SalesOrderHeaders
        where s.SalesOrderNumber == salesOrderNumber
        select s;

    Console.WriteLine("Sales info-");
    foreach (SalesOrderHeader sale in salesInfo)
    {
        Console.WriteLine("Sales ID: " + sale.SalesOrderID);
        Console.WriteLine("Ship date: " + sale.ShipDate);
    }
}
```

```
Dim salesOrderNumber = "SO43663"
Using context As New AdventureWorksEntities()
    Dim salesInfo = _
        From s In context.SalesOrderHeaders _
        Where s.SalesOrderNumber = salesOrderNumber _
        Select s

    Console.WriteLine("Sales info-")
    For Each sale As SalesOrderHeader In salesInfo
        Console.WriteLine("Sales ID: " & sale.SalesOrderID)
        Console.WriteLine("Ship date: " & sale.ShipDate)
    Next
End Using
```

The following example in method-based query syntax shows a query that returns results where the sales order number is equal to "SO43663":

```
    string salesOrderNumber = "SO43663";
    IQueryable<SalesOrderHeader> salesInfo =
        context.SalesOrderHeaders
        .Where(s => s.SalesOrderNumber == salesOrderNumber)
        .Select(s => s);

    Console.WriteLine("Sales info-");
    foreach (SalesOrderHeader sale in salesInfo)
    {
        Console.WriteLine("Sales ID: " + sale.SalesOrderID);
        Console.WriteLine("Ship date: " + sale.ShipDate);
    }
}
```

```
Dim salesOrderNumber = "SO43663"
Using context As New AdventureWorksEntities()
    Dim salesInfo = _
        context.SalesOrderHeaders _
            .Where(Function(s) s.SalesOrderNumber = salesOrderNumber) _
            .Select(Function(s) s)

    Console.WriteLine("Sales info-")
    For Each sale As SalesOrderHeader In salesInfo
        Console.WriteLine("Sales ID: " & sale.SalesOrderID)
        Console.WriteLine("Ship date: " & sale.ShipDate)
    Next
End Using
```

The following example in query expression syntax shows a query that returns sales order information where the ship date is equal to July 8, 2001:

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    DateTime dt = new DateTime(2001, 7, 8);

    IQueryable<SalesOrderHeader> salesInfo =
        from s in context.SalesOrderHeaders
        where s.ShipDate == dt
        select s;

    Console.WriteLine("Orders shipped on August 7, 2001:");
    foreach (SalesOrderHeader sale in salesInfo)
    {
        Console.WriteLine("Sales ID: " + sale.SalesOrderID);
        Console.WriteLine("Total due: " + sale.TotalDue);
        Console.WriteLine();
    }
}
```

```
Using context As New AdventureWorksEntities()
    Dim dt As DateTime = New DateTime(2001, 7, 8)
    Dim salesInfo = _
        From s In context.SalesOrderHeaders _
        Where s.ShipDate = dt _
        Select s

    Console.WriteLine("Orders shipped on August 7, 2001:")
    For Each sale As SalesOrderHeader In salesInfo
        Console.WriteLine("Sales ID: " & sale.SalesOrderID)
        Console.WriteLine("Total due: " & sale.TotalDue)
        Console.WriteLine()
    Next
End Using
```

The following example in method-based query syntax shows a query that returns sales order information where the ship date is equal to July 8, 2001:

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    DateTime dt = new DateTime(2001, 7, 8);

    IQueryable<SalesOrderHeader> salesInfo =
        context.SalesOrderHeaders
        .Where(s => s.ShipDate == dt)
        .Select(s => s);

    Console.WriteLine("Orders shipped on August 7, 2001:");
    foreach (SalesOrderHeader sale in salesInfo)
    {
        Console.WriteLine("Sales ID: " + sale.SalesOrderID);
        Console.WriteLine("Total due: " + sale.TotalDue);
        Console.WriteLine();
    }
}
```

```
Using context As New AdventureWorksEntities()
    Dim dt As DateTime = New DateTime(2001, 7, 8)

    Dim salesInfo = _
        context.SalesOrderHeaders _
        .Where(Function(s) s.ShipDate = dt) _
        .Select(Function(s) s)

    Console.WriteLine("Orders shipped on August 7, 2001:")
    For Each sale As SalesOrderHeader In salesInfo
        Console.WriteLine("Sales ID: " & sale.SalesOrderID)
        Console.WriteLine("Total due: " & sale.TotalDue)
        Console.WriteLine()
    Next
End Using
```

Expressions that yield a constant are converted at the server, and no attempt to do local evaluation is performed. The following example uses an expression in the `Where` clause that yields a constant.

```
Decimal totalDue = 200 + 3;
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<string> salesInfo =
        from s in context.SalesOrderHeaders
        where s.TotalDue >= totalDue
        select s.SalesOrderNumber;

    Console.WriteLine("Sales order numbers:");
    foreach (string orderNum in salesInfo)
    {
        Console.WriteLine(orderNum);
    }
}
```

```
Dim totalDue = 200 + 3
Using context As New AdventureWorksEntities()
    Dim salesInfo = _
        From s In context.SalesOrderHeaders _
        Where s.TotalDue >= totalDue _
        Select s.SalesOrderNumber

    Console.WriteLine("Sales order numbers:")
    For Each orderNum As String In salesInfo
        Console.WriteLine(orderNum)
    Next
End Using
```

LINQ to Entities does not support using a user class as a constant. However, a property reference on a user class is considered a constant, and will be converted to a command tree constant expression and executed on the data source.

```
class AClass { public int ID;}
```

```
Class AClass
    Public ID As Integer
End Class
```

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    AClass aClass = new AClass();
    aClass.ID = 43663;

    IQueryable<SalesOrderHeader> salesInfo =
        from s in context.SalesOrderHeaders
        where s.SalesOrderID == aClass.ID
        select s;

    Console.WriteLine("Order info-");
    foreach (SalesOrderHeader sale in salesInfo)
    {
        Console.WriteLine("Sales order number: " + sale.SalesOrderNumber);
        Console.WriteLine("Total due: " + sale.TotalDue);
        Console.WriteLine();
    }
}
```

```
Using context As New AdventureWorksEntities()
    Dim aClass As AClass = New aClass()
    aClass.ID = 43663

    Dim salesInfo = _
        From s In context.SalesOrderHeaders _
        Where s.SalesOrderID = aClass.ID _
        Select s

    Console.WriteLine("Order info-")
    For Each sale As SalesOrderHeader In salesInfo
        Console.WriteLine("Sales order number: " & sale.SalesOrderNumber)
        Console.WriteLine("Total due: " & sale.TotalDue)
        Console.WriteLine()
    Next
End Using
```

Methods that return a constant expression are not supported. The following example contains a method in the `Where` clause that returns a constant. This example will throw an exception at run time.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    MyClass2 myClass = new MyClass2();

    //Throws a NotSupportedException
    IQueryable<SalesOrderHeader> salesInfo =
        from s in context.SalesOrderHeaders
        where s.SalesOrderID == myClass.returnInt()
        select s;

    Console.WriteLine("Order info-");
    try
    {
        foreach (SalesOrderHeader sale in salesInfo)
        {
            Console.WriteLine("Sales order number: " + sale.SalesOrderNumber);
            Console.WriteLine("Total due: " + sale.TotalDue);
            Console.WriteLine();
        }
    }
    catch (NotSupportedException ex)
    {
        Console.WriteLine("Exception: {0}", ex.Message);
    }
}
```

```
Using context As New AdventureWorksEntities()
    Dim aClass2 As AClass2 = New aClass2()

    ' Throws a NotSupportedException.
    Dim salesInfo = _
        From s In context.SalesOrderHeaders _
        Where s.SalesOrderID = aClass2.returnInt() _
        Select s

    Console.WriteLine("Order info-")
    Try
        For Each sale As SalesOrderHeader In salesInfo
            Console.WriteLine("Sales order number: " & sale.SalesOrderNumber)
            Console.WriteLine("Total due: " & sale.TotalDue)
            Console.WriteLine()
        Next
    Catch ex As NotSupportedException
        Console.WriteLine("Exception: {0}", ex.Message)
    End Try
End Using
```

See also

- Expressions in LINQ to Entities Queries

Null Comparisons

11/8/2022 • 3 minutes to read • Edit Online

A `null` value in the data source indicates that the value is unknown. In LINQ to Entities queries, you can check for null values so that certain calculations or comparisons are only performed on rows that have valid, or non-null, data. CLR null semantics, however, may differ from the null semantics of the data source. Most databases use a version of three-valued logic to handle null comparisons. That is, a comparison against a null value does not evaluate to `true` or `false`, it evaluates to `unknown`. Often this is an implementation of ANSI nulls, but this is not always the case.

By default in SQL Server, the null-equals-null comparison returns a null value. In the following example, the rows where `ShipDate` is null are excluded from the result set, and the Transact-SQL statement would return 0 rows.

```
-- Find order details and orders with no ship date.
SELECT h.SalesOrderID
FROM Sales.SalesOrderHeader h
JOIN Sales.SalesOrderDetail o ON o.SalesOrderID = h.SalesOrderID
WHERE h.ShipDate IS Null
```

This is very different from the CLR null semantics, where the null-equals-null comparison returns true.

The following LINQ query is expressed in the CLR, but it is executed in the data source. Because there is no guarantee that CLR semantics will be honored at the data source, the expected behavior is indeterminate.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<SalesOrderHeader> orders = context.SalesOrderHeaders;
    ObjectSet<SalesOrderDetail> details = context.SalesOrderDetails;

    var query =
        from order in orders
        join detail in details
        on order.SalesOrderID
        equals detail.SalesOrderID
        where order.ShipDate == null
        select order.SalesOrderID;

    foreach (var OrderID in query)
    {
        Console.WriteLine("OrderID : {0}", OrderID);
    }
}
```

```
Using context As New AdventureWorksEntities()

    Dim orders As ObjectSet(Of SalesOrderHeader) = context.SalesOrderHeaders
    Dim details As ObjectSet(Of SalesOrderDetail) = context.SalesOrderDetails

    Dim query = _
        From order In orders _
        Join detail In details _
        On order.SalesOrderID _
        Equals detail.SalesOrderID _
        Where order.ShipDate = Nothing
        Select order.SalesOrderID


    For Each orderID In query
        Console.WriteLine("OrderID: {0} ", orderID)
    Next
End Using
```

Key Selectors

A *key selector* is a function used in the standard query operators to extract a key from an element. In the key selector function, an expression can be compared with a constant. CLR null semantics are exhibited if an expression is compared to a null constant or if two null constants are compared. Store null semantics are exhibited if two columns with null values in the data source are compared. Key selectors are found in many of the grouping and ordering standard query operators, such as GroupBy, and are used to select keys by which to order or group the query results.

Null Property on a Null Object

In the Entity Framework, the properties of a null object are null. When you attempt to reference a property of a null object in the CLR, you will receive a NullReferenceException. When a LINQ query involves a property of a null object, this can result in inconsistent behavior.

For example, in the following query, the cast to `NewProduct` is done in the command tree layer, which might result in the `Introduced` property being null. If the database defined null comparisons such that the DateTime comparison evaluates to true, the row will be included.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{

    DateTime dt = new DateTime();
    var query = context.Products
        .Where(p => (p as NewProduct).Introduced > dt)
        .Select(x => x);
}
```

```
Using context As New AdventureWorksEntities()
    Dim dt As DateTime = New DateTime()
    Dim query = context.Products _
        .Where(Function(p) _
            ((DirectCast(p, NewProduct)).Introduced > dt)) _
        .Select(Function(x) x)
End Using
```

Passing Null Collections to Aggregate Functions

In LINQ to Entities, when you pass a collection that supports `IQueryable` to an aggregate function, aggregate operations are performed at the database. There might be differences in the results of a query that was performed in-memory and a query that was performed at the database. With an in-memory query, if there are no matches, the query returns zero. At the database, the same query returns `null`. If a `null` value is passed to a LINQ aggregate function, an exception will be thrown. To accept possible `null` values, cast the types and the properties of the types that receive query results to nullable value types.

See also

- Expressions in LINQ to Entities Queries

Initialization Expressions

11/8/2022 • 3 minutes to read • Edit Online

An initialization expression initializes a new object. Most initialization expressions are supported, including most new C# 3.0 and Visual Basic 9.0 initialization expressions. The following types can be initialized and returned by a LINQ to Entities query:

- A collection of zero or more typed entity objects or a projection of complex types that are defined in the conceptual model.
- CLR types supported by the Entity Framework.
- Inline collections.
- Anonymous types.

Anonymous type initialization is shown in the following example in query expression syntax:

```
Decimal totalDue = 200;
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var salesInfo =
        from s in context.SalesOrderHeaders
        where s.TotalDue >= totalDue
        select new { s.SalesOrderNumber, s.TotalDue };

    Console.WriteLine("Sales order numbers:");
    foreach (var sale in salesInfo)
    {
        Console.WriteLine("Order number: " + sale.SalesOrderNumber);
        Console.WriteLine("Total due: " + sale.TotalDue);
        Console.WriteLine("");
    }
}
```

```
Dim totalDue = 200
Using context As New AdventureWorksEntities()
    Dim salesInfo = _
        From s In context.SalesOrderHeaders _
        Where s.TotalDue >= totalDue _
        Select New With {s.SalesOrderNumber, s.TotalDue}

    Console.WriteLine("Sales order numbers:")
    For Each sale In salesInfo
        Console.WriteLine("Order number: " & sale.SalesOrderNumber)
        Console.WriteLine("Total due: " & sale.TotalDue)
        Console.WriteLine("")
    Next
End Using
```

The following example in method-based query syntax shows anonymous type initialization:

```
Decimal totalDue = 200;
using (AdventureWorksEntities context = new AdventureWorksEntities())
{

    var salesInfo =
        context.SalesOrderHeaders
        .Where(s => s.TotalDue >= totalDue)
        .Select(s => new { s.SalesOrderNumber, s.TotalDue });

    Console.WriteLine("Sales order numbers:");
    foreach (var sale in salesInfo)
    {
        Console.WriteLine("Order number: " + sale.SalesOrderNumber);
        Console.WriteLine("Total due: " + sale.TotalDue);
        Console.WriteLine("");
    }
}
```

```
Dim totalDue = 200
Using context As New AdventureWorksEntities()

    Dim salesInfo = _
        context.SalesOrderHeaders _
        .Where(Function(s) s.TotalDue >= totalDue) _
        .Select(Function(s) New With {s.SalesOrderNumber, s.TotalDue})

    Console.WriteLine("Sales order numbers:")
    For Each sale In salesInfo
        Console.WriteLine("Order number: " & sale.SalesOrderNumber)
        Console.WriteLine("Total due: " & sale.TotalDue)
        Console.WriteLine("")
    Next
End Using
```

User-defined class initialization is also supported. The C# 3.0 and Visual Basic 9.0 initialization pattern is supported and assumes that the property getter and setter are symmetric. The following example in query expression syntax shows a custom class being initialized in the query:

```
class MyOrder { public string SalesOrderNumber; public DateTime? ShipDate; }
```

```
Class MyOrder
    Public SalesOrderNumber As String
    Public ShipDate As DateTime?
End Class
```

```
Decimal totalDue = 200;
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<MyOrder> salesInfo =
        from s in context.SalesOrderHeaders
        where s.TotalDue >= totalDue
        select new MyOrder
        {
            SalesOrderNumber = s.SalesOrderNumber,
            ShipDate = s.ShipDate
        };

    Console.WriteLine("Sales order info:");
    foreach (MyOrder order in salesInfo)
    {
        Console.WriteLine("Order number: " + order.SalesOrderNumber);
        Console.WriteLine("Ship date: " + order.ShipDate);
        Console.WriteLine("");
    }
}
```

```
Dim totalDue = 200
Using context As New AdventureWorksEntities()
    Dim salesInfo = _
        From s In context.SalesOrderHeaders _
        Where s.TotalDue >= totalDue _
        Select New MyOrder With _
               { _
                   .SalesOrderNumber = s.SalesOrderNumber, _
                   .ShipDate = s.ShipDate _
               }

    Console.WriteLine("Sales order info:")
    For Each order As MyOrder In salesInfo
        Console.WriteLine("Order number: " & order.SalesOrderNumber)
        Console.WriteLine("Ship date: " & order.ShipDate)
        Console.WriteLine("")
    Next
End Using
```

The following example in method-based query syntax shows a custom class being initialized in the query:

```
Decimal totalDue = 200;
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<MyOrder> salesInfo =
        context.SalesOrderHeaders
        .Where(s => s.TotalDue >= totalDue)
        .Select(s => new MyOrder
        {
            SalesOrderNumber = s.SalesOrderNumber,
            ShipDate = s.ShipDate
        });

    Console.WriteLine("Sales order info:");
    foreach (MyOrder order in salesInfo)
    {
        Console.WriteLine("Order number: " + order.SalesOrderNumber);
        Console.WriteLine("Ship date: " + order.ShipDate);
        Console.WriteLine("");
    }
}
```

```
Dim totalDue = 200
Using context As New AdventureWorksEntities()

    Dim salesInfo As IQueryable(Of MyOrder) = _
        context.SalesOrderHeaders _
        .Where(Function(s) s.TotalDue >= totalDue) _
        .Select(Function(s) New MyOrder With _
               { _
                   .SalesOrderNumber = s.SalesOrderNumber, _
                   .ShipDate = s.ShipDate _
               })

    Console.WriteLine("Sales order info:")
    For Each order As MyOrder In salesInfo
        Console.WriteLine("Order number: " & order.SalesOrderNumber)
        Console.WriteLine("Ship date: " & order.ShipDate)
        Console.WriteLine("")
    Next
End Using
```

See also

- Expressions in LINQ to Entities Queries

Calling Functions in LINQ to Entities Queries

11/8/2022 • 2 minutes to read • Edit Online

The topics in this section describe how to call functions in LINQ to Entities queries.

The EntityFunctions and SqlFunctions classes provide access to canonical and database functions as part of the Entity Framework. For more information, see How to: Call Canonical Functions and How to: Call Database Functions.

The process for calling a custom function requires three basic steps:

1. Define a function in your conceptual model or declare a function in your storage model.
2. Add a method to your application and map it to the function in the model with an EdmFunctionAttribute.
3. Call the function in a LINQ to Entities query.

For more information, see the topics in this section.

In This Section

How to: Call Canonical Functions

How to: Call Database Functions

How to: Call Custom Database Functions

How to: Call Model-Defined Functions in Queries

How to: Call Model-Defined Functions as Object Methods

See also

- Queries in LINQ to Entities
- Canonical Functions
- .edmx File Overview
- How to: Define Custom Functions in the Conceptual Model

How to: Call Canonical Functions

11/8/2022 • 2 minutes to read • Edit Online

The EntityFunctions class contains methods that expose canonical functions to use in LINQ to Entities queries. For information about canonical functions, see Canonical Functions.

> **NOTE**
>
> The AsUnicode and AsNonUnicode methods in the EntityFunctions class do not have canonical function equivalents.

Canonical functions that perform a calculation on a set of values and return a single value (also known as aggregate canonical functions) can be directly invoked. Other canonical functions can only be called as part of a LINQ to Entities query. To call an aggregate function directly, you must pass an ObjectQuery<T> to the function. For more information, see the second example below.

You can call some canonical functions by using common language runtime (CLR) methods in LINQ to Entities queries. For a list of CLR methods that map to canonical functions, see CLR Method to Canonical Function Mapping.

Example 1

The following example uses the AdventureWorks Sales Model. The example executes a LINQ to Entities query that uses the DiffDays method to return all products for which the difference between `SellEndDate` and `SellStartDate` is less than 365 days:

```
using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
{
    var products = from p in AWEntities.Products
                   where EntityFunctions.DiffDays(p.SellEndDate, p.SellStartDate) < 365
                   select p;
    foreach (var product in products)
    {
        Console.WriteLine(product.ProductID);
    }
}
```

```
Using AWEntities As New AdventureWorksEntities()
    Dim products = From p In AWEntities.Products _
                   Where EntityFunctions.DiffDays(p.SellEndDate, p.SellStartDate) < 365 _
                   Select p

    For Each product In products
        Console.WriteLine(product.ProductID)
    Next
End Using
```

Example 2

The following example uses the AdventureWorks Sales Model. The example calls the aggregate StandardDeviation method directly to return the standard deviation of `SalesOrderHeader` subtotals. Note that an ObjectQuery<T> is passed to the function, which allows it to be called without being part of a LINQ to Entities

query.

```
using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
{
    double? stdDev = EntityFunctions.StandardDeviation(
        from o in AWEntities.SalesOrderHeaders
        select o.SubTotal);

    Console.WriteLine(stdDev);
}
```

```
Using AWEntities As New AdventureWorksEntities()
    Dim stdDev As Double? = EntityFunctions.StandardDeviation( _
        From o In AWEntities.SalesOrderHeaders _
        Select o.SubTotal)

    Console.WriteLine(stdDev)
End Using
```

See also

- Calling Functions in LINQ to Entities Queries
- Queries in LINQ to Entities

How to: Call Database Functions

11/8/2022 • 2 minutes to read • Edit Online

The SqlFunctions class contains methods that expose SQL Server functions to use in LINQ to Entities queries. When you use SqlFunctions methods in LINQ to Entities queries, the corresponding database functions are executed in the database.

> **NOTE**
>
> Database functions that perform a calculation on a set of values and return a single value (also known as aggregate database functions) can be directly invoked. Other canonical functions can only be called as part of a LINQ to Entities query. To call an aggregate function directly, you must pass an ObjectQuery<T> to the function. For more information, see the second example below.

> **NOTE**
>
> The methods in the SqlFunctions class are specific to SQL Server functions. Similar classes that expose database functions may be available through other providers.

Example 1

The following example uses the AdventureWorks Sales Model. The example executes a LINQ to Entities query that uses the CharIndex method to return all contacts whose last name starts with "Si":

```
using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
{
    // SqlFunctions.CharIndex is executed in the database.
    var contacts = from c in AWEntities.Contacts
                   where SqlFunctions.CharIndex("Si", c.LastName) == 1
                   select c;

    foreach (var contact in contacts)
    {
        Console.WriteLine(contact.LastName);
    }
}
```

```
Using AWEntities As New AdventureWorksEntities()

    ' SqlFunctions.CharIndex is executed in the database.
    Dim contacts = From c In AWEntities.Contacts _
                   Where SqlFunctions.CharIndex("Si", c.LastName) = 1 _
                   Select c

    For Each contact In contacts
        Console.WriteLine(contact.LastName)
    Next
End Using
```

Example 2

The following example uses the AdventureWorks Sales Model. The example calls the aggregate ChecksumAggregate method directly. Note that an ObjectQuery<T> is passed to the function, which allows it to be called without being part of a LINQ to Entities query.

```
using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
{
    // SqlFunctions.ChecksumAggregate is executed in the database.
    decimal? checkSum = SqlFunctions.ChecksumAggregate(
        from o in AWEntities.SalesOrderHeaders
        select o.SalesOrderID);

    Console.WriteLine(checkSum);
}
```

```
Using AWEntities As New AdventureWorksEntities()

    ' SqlFunctions.ChecksumAggregate is executed in the database.
    Dim checkSum As Integer = SqlFunctions.ChecksumAggregate( _
        From o In AWEntities.SalesOrderHeaders _
        Select o.SalesOrderID)

    Console.WriteLine(checkSum)
End Using
```

See also

- Calling Functions in LINQ to Entities Queries
- Queries in LINQ to Entities

How to: Call Custom Database Functions

11/8/2022 • 2 minutes to read • Edit Online

This topic describes how to call custom functions that are defined in the database from within LINQ to Entities queries.

Database functions that are called from LINQ to Entities queries are executed in the database. Executing functions in the database can improve application performance.

The procedure below provides a high-level outline for calling a custom database function. The example that follows provides more detail about the steps in the procedure.

To call custom functions that are defined in the database

1. Create a custom function in your database.

   For more information about creating custom functions in SQL Server, see CREATE FUNCTION (Transact-SQL).

2. Declare a function in the store schema definition language (SSDL) of your .edmx file. The name of the function must be the same as the name of the function declared in the database.

   For more information, see Function Element (SSDL).

3. Add a corresponding method to a class in your application code and apply a EdmFunctionAttribute to the method Note that the NamespaceName and FunctionName parameters of the attribute are the namespace name of the conceptual model and the function name in the conceptual model respectively. Function name resolution for LINQ is case sensitive.

4. Call the method in a LINQ to Entities query.

Example 1

The following example demonstrates how to call a custom database function from within a LINQ to Entities query. The example uses the School model. For information about the School model, see Creating the School Sample Database and Generating the School .edmx File.

The following code adds the `AvgStudentGrade` function to the School sample database.

> **NOTE**
>
> The steps for calling a custom database function are the same regardless of the database server. However, the code below is specific to creating a function in a SQL Server database. The code for creating a custom function in other database servers might differ.

```
USE [School]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE FUNCTION [dbo].[AvgStudentGrade](@studentId INT)
RETURNS DECIMAL(3,2)
AS
    BEGIN
    DECLARE @avg DECIMAL(3,2);
    SELECT @avg = avg(Grade) FROM StudentGrade WHERE StudentID = @studentId;

    RETURN @avg;
END
```

Example 2

Next, declare a function in the store schema definition language (SSDL) of your *.edmx* file. The following code declares the `AvgStudentGrade` function in SSDL:

```
<Function Name="AvgStudentGrade" ReturnType="decimal" Schema="dbo" >
  <Parameter Name="studentId" Mode="In" Type="int" />
</Function>
```

Example 3

Now, create a method and map it to the function declared in the SSDL. The method in the following class is mapped to the function defined in the SSDL (above) by using an EdmFunctionAttribute. When this method is called, the corresponding function in the database is executed.

```
[EdmFunction("SchoolModel.Store", "AvgStudentGrade")]
public static decimal? AvgStudentGrade(int studentId)
{
    throw new NotSupportedException("Direct calls are not supported.");
}
```

```
<EdmFunction("SchoolModel.Store", "AvgStudentGrade")>
Public Function AvgStudentGrade(ByVal studentId As Integer) _
    As Nullable(Of Decimal)
    Throw New NotSupportedException("Direct calls are not supported.")
End Function
```

Example 4

Finally, call the method in a LINQ to Entities query. The following code displays students' last names and average grades to the console:

```
using (SchoolEntities context = new SchoolEntities())
{
    var students = from s in context.People
                   where s.EnrollmentDate != null
                   select new
                   {
                       name = s.LastName,
                       avgGrade = AvgStudentGrade(s.PersonID)
                   };

    foreach (var student in students)
    {
        Console.WriteLine("{0}: {1}", student.name, student.avgGrade);
    }
}
```

```
Using context As New SchoolEntities()
    Dim students = From s In context.People _
                   Where s.EnrollmentDate IsNot Nothing _
                   Select New With {.name = s.LastName, _
                                    .avgGrade = AvgStudentGrade(s.PersonID)}

    For Each student In students
        Console.WriteLine("{0}: {1}", _
                          student.name, _
                          student.avgGrade)
    Next
End Using
```

See also

- .edmx File Overview
- Queries in LINQ to Entities

How to: Call Model-Defined Functions in Queries

11/8/2022 • 2 minutes to read • Edit Online

This topic describes how to call functions that are defined in the conceptual model from within LINQ to Entities queries.

The procedure below provides a high-level outline for calling a model-defined function from within a LINQ to Entities query. The example that follows provides more detail about the steps in the procedure. The procedure assumes that you have defined a function in the conceptual model. For more information, see How to: Define Custom Functions in the Conceptual Model.

To call a function defined in the conceptual model

1. Add a common language runtime (CLR) method to your application that maps to the function defined in the conceptual model. To map the method, you must apply an EdmFunctionAttribute to the method. Note that the NamespaceName and FunctionName parameters of the attribute are the namespace name of the conceptual model and the function name in the conceptual model respectively. Function name resolution for LINQ is case sensitive.

2. Call the function in a LINQ to Entities query.

Example 1

The following example demonstrates how to call a function that is defined in the conceptual model from within a LINQ to Entities query. The example uses the School model. For information about the School model, see Creating the School Sample Database and Generating the School .edmx File.

The following conceptual model function returns the number of years since an instructor was hired. For information about adding the function to a conceptual model, see How to: Define Custom Functions in the Conceptual Model.)

```
<Function Name="YearsSince" ReturnType="Edm.Int32">
  <Parameter Name="date" Type="Edm.DateTime" />
  <DefiningExpression>
    Year(CurrentDateTime()) - Year(date)
  </DefiningExpression>
</Function>
```

Example 2

Next, add the following method to your application and use an EdmFunctionAttribute to map it to the conceptual model function:

```
[EdmFunction("SchoolModel", "YearsSince")]
public static int YearsSince(DateTime date)
{
    throw new NotSupportedException("Direct calls are not supported.");
}
```

```
<EdmFunction("SchoolModel", "YearsSince")>
Public Function YearsSince(ByVal date1 As DateTime) _
    As Integer
    Throw New NotSupportedException("Direct calls are not supported.")
End Function
```

Example 3

Now you can call the conceptual model function from within a LINQ to Entities query. The following code calls the method to display all instructors that were hired more than ten years ago:

```
using (SchoolEntities context = new SchoolEntities())
{
    // Retrieve instructors hired more than 10 years ago.
    var instructors = from p in context.People
                      where YearsSince((DateTime)p.HireDate) > 10
                      select p;

    foreach (var instructor in instructors)
    {
        Console.WriteLine(instructor.LastName);
    }
}
```

```
Using context As New SchoolEntities()
    ' Retrieve instructors hired more than 10 years ago.
    Dim instructors = From p In context.People _
                      Where YearsSince(CType(p.HireDate, DateTime?)) > 10 _
                      Select p

    For Each instructor In instructors
        Console.WriteLine(instructor.LastName)
    Next
End Using
```

See also

- .edmx File Overview
- Queries in LINQ to Entities
- Calling Functions in LINQ to Entities Queries
- How to: Call Model-Defined Functions as Object Methods

How to: Call Model-Defined Functions as Object Methods

11/8/2022 • 6 minutes to read • Edit Online

This topic describes how to call a model-defined function as a method on an ObjectContext object or as a static method on a custom class. A *model-defined function* is a function that is defined in the conceptual model. The procedures in the topic describe how to call these functions directly instead of calling them from LINQ to Entities queries. For information about calling model-defined functions in LINQ to Entities queries, see How to: Call Model-Defined Functions in Queries.

Whether you call a model-defined function as an ObjectContext method or as a static method on a custom class, you must first map the method to the model-defined function with an EdmFunctionAttribute. However, when you define a method on the ObjectContext class, you must use the QueryProvider property to expose the LINQ provider, whereas when you define a static method on a custom class, you must use the Provider property to expose the LINQ provider. For more information, see the examples that follow the procedures below.

The procedures below provide high-level outlines for calling a model-defined function as a method on an ObjectContext object and as a static method on a custom class. The examples that follow provide more detail about the steps in the procedures. The procedures assume that you have defined a function in the conceptual model. For more information, see How to: Define Custom Functions in the Conceptual Model.

To call a model-defined function as a method on an ObjectContext object

1. Add a source file to extend the partial class derived from the ObjectContext class, auto-generated by the Entity Framework tools. Defining the CLR stub in a separate source file will prevent the changes from being lost when the file is regenerated.
2. Add a common language runtime (CLR) method to your ObjectContext class that does the following:
   - Maps to the function defined in the conceptual model. To map the method, you must apply an EdmFunctionAttribute to the method. Note that the NamespaceName and FunctionName parameters of the attribute are the namespace name of the conceptual model and the function name in the conceptual model, respectively. Function name resolution for LINQ is case sensitive.
   - Returns the results of the Execute method that is returned by the QueryProvider property.
3. Call the method as a member on an instance of the ObjectContext class.

To call a model-defined function as static method on a custom class

1. Add a class to your application with a static method that does the following:
   - Maps to the function defined in the conceptual model. To map the method, you must apply an EdmFunctionAttribute to the method. Note that the NamespaceName and FunctionName parameters of the attribute are the namespace name of the conceptual model and the function name in the conceptual model, respectively.
   - Accepts an IQueryable argument.
   - Returns the results of the Execute method that is returned by the Provider property.

2. Call the method as a member a static method on the custom class

Example 1

Calling a Model-Defined Function as a Method on an ObjectContext Object

The following example demonstrates how to call a model-defined function as a method on an ObjectContext object. The example uses the AdventureWorks Sales Model.

Consider the conceptual model function below that returns product revenue for a specified product. (For information about adding the function to your conceptual model, see How to: Define Custom Functions in the Conceptual Model.)

```
<Function Name="GetProductRevenue" ReturnType="Edm.Decimal">
  <Parameter Name="productID" Type="Edm.Int32" />
  <DefiningExpression>
    SUM( SELECT VALUE((s.UnitPrice - s.UnitPriceDiscount)  * s.OrderQty)
    FROM AdventureWorksEntities.SalesOrderDetails as s
    WHERE s.ProductID = productID)
  </DefiningExpression>
</Function>
```

Example 2

The following code adds a method to the `AdventureWorksEntities` class that maps to the conceptual model function above.

```
public partial class AdventureWorksEntities : ObjectContext
{
    [EdmFunction("AdventureWorksModel", "GetProductRevenue")]
    public decimal? GetProductRevenue(int productId)
    {
        return this.QueryProvider.Execute<decimal?>(Expression.Call(
            Expression.Constant(this),
            (MethodInfo)MethodInfo.GetCurrentMethod(),
            Expression.Constant(productId, typeof(int))));
    }
}
```

```
Partial Public Class AdventureWorksEntities
    Inherits ObjectContext

    <EdmFunction("AdventureWorksModel", "GetProductRevenue")>
    Public Function GetProductRevenue(ByVal details As _
                    IQueryable(Of SalesOrderDetail)) As _
                    System.Nullable(Of Decimal)
        Return Me.QueryProvider.Execute(Of System.Nullable(Of Decimal)) _
            (Expression.[Call](Expression.Constant(Me), _
            DirectCast(MethodInfo.GetCurrentMethod(), MethodInfo), _
            Expression.Constant(details, GetType(IQueryable(Of SalesOrderDetail)))))
    End Function
End Class
```

Example 3

The following code calls the method above to display the product revenue for a specified product:

```
using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
{
    int productId = 776;

    Console.WriteLine(AWEntities.GetProductRevenue(productId));
}
```

```
Using AWEntities As New AdventureWorksEntities()

    Dim productId As Integer = 776

    Dim details = From s In AWEntities.SalesOrderDetails _
                  Where s.ProductID = productId _
                  Select s

    Console.WriteLine(AWEntities.GetProductRevenue(details))
End Using
```

Example 4

The following example demonstrates how to call a model-defined function that returns a collection (as an IQueryable<T> object). Consider the conceptual model function below that returns all the `SalesOrderDetails` for a given product ID.

```
<Function Name="GetDetailsById"
              ReturnType="Collection(AdventureWorksModel.SalesOrderDetail)">
      <Parameter Name="productID" Type="Edm.Int32" />
      <DefiningExpression>
        SELECT VALUE s
        FROM AdventureWorksEntities.SalesOrderDetails AS s
        WHERE s.ProductID = productID
      </DefiningExpression>
    </Function>
```

Example 5

The following code adds a method to the `AdventureWorksEntities` class that maps to the conceptual model function above.

```
public partial class AdventureWorksEntities : ObjectContext
{
    [EdmFunction("AdventureWorksModel", "GetDetailsById")]
    public IQueryable<SalesOrderDetail> GetDetailsById(int productId)
    {
        return this.QueryProvider.CreateQuery<SalesOrderDetail>(Expression.Call(
            Expression.Constant(this),
            (MethodInfo)MethodInfo.GetCurrentMethod(),
            Expression.Constant(productId, typeof(int))));
    }
}
```

```
Partial Public Class AdventureWorksEntities
    Inherits ObjectContext
    <EdmFunction("AdventureWorksModel", "GetDetailsById")> _
    Public Function GetDetailsById(ByVal productId As Integer) _
            As IQueryable(Of SalesOrderDetail)
        Return Me.QueryProvider.CreateQuery(Of SalesOrderDetail) _
            (Expression.[Call](Expression.Constant(Me), _
             DirectCast(MethodInfo.GetCurrentMethod(), MethodInfo), _
             Expression.Constant(productId, GetType(Integer))))
    End Function
End Class
```

Example 6

The following code calls the method. Note that the returned IQueryable<T> query is further refined to return line totals for each `SalesOrderDetail`.

```
using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
{
    int productId = 776;

    var lineTotals = AWEntities.GetDetailsById(productId).Select(d =>d.LineTotal);

    foreach(var lineTotal in lineTotals)
    {
        Console.WriteLine(lineTotal);
    }
}
```

```
Using AWEntities As New AdventureWorksEntities()
    Dim productId As Integer = 776

    Dim lineTotals = AWEntities.GetDetailsById(productId).[Select](Function(d) d.LineTotal)

    For Each lineTotal In lineTotals
        Console.WriteLine(lineTotal)
    Next
```

Example 7

Calling a Model-Defined Function as a Static Method on a Custom Class

The next example demonstrates how to call a model-defined function as a static method on a custom class. The example uses the AdventureWorks Sales Model.

> **NOTE**
> When you call a model-defined function as a static method on a custom class, the model-defined function must accept a collection and return an aggregation of values in the collection.

Consider the conceptual model function below that returns product revenue for a SalesOrderDetail collection. (For information about adding the function to your conceptual model, see How to: Define Custom Functions in the Conceptual Model.).

```
<Function Name="GetProductRevenue" ReturnType="Edm.Decimal">
  <Parameter Name="details"
             Type="Collection(AdventureWorksModel.SalesOrderDetail)" />
  <DefiningExpression>
    SUM( SELECT VALUE((s.UnitPrice - s.UnitPriceDiscount)  * s.OrderQty)
    FROM details as s)
  </DefiningExpression>
</Function>
```

Example 8

The following code adds a class to your application that contains a static method that maps to the conceptual model function above.

```
public class MyClass
{
    [EdmFunction("AdventureWorksModel", "GetProductRevenue")]
    public static decimal? GetProductRevenue(IQueryable<SalesOrderDetail> details)
    {
        return details.Provider.Execute<decimal?>(Expression.Call(
            (MethodInfo)MethodInfo.GetCurrentMethod(),
            Expression.Constant(details, typeof(IQueryable<SalesOrderDetail>))));
    }
}
```

```
Public Class [MyClass]
    <EdmFunction("AdventureWorksModel", "GetProductRevenue")> _
    Public Shared Function GetProductRevenue(ByVal details As _
                IQueryable(Of SalesOrderDetail)) As _
                System.Nullable(Of Decimal)
        Return details.Provider.Execute(Of System.Nullable(Of Decimal)) _
            (Expression.[Call](DirectCast(MethodInfo.GetCurrentMethod(), MethodInfo), _
            Expression.Constant(details, GetType(IQueryable(Of SalesOrderDetail)))))
    End Function
End Class
```

Example 9

The following code calls the method above to display the product revenue for a SalesOrderDetail collection:

```
using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
{
    int productId = 776;

    var details = from s in AWEntities.SalesOrderDetails
                  where s.ProductID == productId select s;

    Console.WriteLine(MyClass.GetProductRevenue(details));
}
```

```
Using AWEntities As New AdventureWorksEntities()
    Dim productId As Integer = 776

    Dim details = From s In AWEntities.SalesOrderDetails _
                  Where s.ProductID = productId _
                  Select s

    Console.WriteLine([MyClass].GetProductRevenue(details))
End Using
```

See also

- .edmx File Overview
- Queries in LINQ to Entities
- Calling Functions in LINQ to Entities Queries

Compiled queries (LINQ to Entities)

11/8/2022 • 7 minutes to read • Edit Online

When you have an application that executes structurally similar queries many times in the Entity Framework, you can frequently increase performance by compiling the query one time and executing it several times with different parameters. For example, an application might have to retrieve all the customers in a particular city; the city is specified at run time by the user in a form. LINQ to Entities supports using compiled queries for this purpose.

Starting with .NET Framework 4.5, LINQ queries are cached automatically. However, you can still use compiled LINQ queries to reduce this cost in later executions and compiled queries can be more efficient than LINQ queries that are automatically cached. LINQ to Entities queries that apply the `Enumerable.Contains` operator to in-memory collections are not automatically cached. Also, parameterizing in-memory collections in compiled LINQ queries is not allowed.

The CompiledQuery class provides compilation and caching of queries for reuse. Conceptually, this class contains a CompiledQuery's `Compile` method with several overloads. Call the `Compile` method to create a new delegate to represent the compiled query. The `Compile` methods, provided with a ObjectContext and parameter values, return a delegate that produces some result (such as an IQueryable<T> instance). The query compiles once during only the first execution. The merge options set for the query at the time of the compilation cannot be changed later. Once the query is compiled, you can only supply parameters of primitive type but you cannot replace parts of the query that would change the generated SQL. For more information, see EF Merge Options and Compiled Queries.

The LINQ to Entities query expression that the CompiledQuery's `Compile` method compiles is represented by one of the generic `Func` delegates, such as Func<T1,T2,T3,T4,TResult>. At most, the query expression can encapsulate an `ObjectContext` parameter, a return parameter, and 16 query parameters. If more than 16 query parameters are required, you can create a structure whose properties represent query parameters. You can then use the properties on the structure in the query expression after you set the properties.

Example 1

The following example compiles and then invokes a query that accepts a Decimal input parameter and returns a sequence of orders where the total due is greater than or equal to $200.00:

```
static readonly Func<AdventureWorksEntities, Decimal, IQueryable<SalesOrderHeader>> s_compiledQuery2 =
    CompiledQuery.Compile<AdventureWorksEntities, Decimal, IQueryable<SalesOrderHeader>>(
            (ctx, total) => from order in ctx.SalesOrderHeaders
                            where order.TotalDue >= total
                            select order);

static void CompiledQuery2()
{
    using (AdventureWorksEntities context = new AdventureWorksEntities())
    {
        Decimal totalDue = 200.00M;

        IQueryable<SalesOrderHeader> orders = s_compiledQuery2.Invoke(context, totalDue);

        foreach (SalesOrderHeader order in orders)
        {
            Console.WriteLine("ID: {0}  Order date: {1} Total due: {2}",
                order.SalesOrderID,
                order.OrderDate,
                order.TotalDue);
        }
    }
}
```

```
ReadOnly s_compQuery2 As Func(Of AdventureWorksEntities, Decimal, IQueryable(Of SalesOrderHeader)) = _
    CompiledQuery.Compile(Of AdventureWorksEntities, Decimal, IQueryable(Of SalesOrderHeader))( _
                Function(ctx As AdventureWorksEntities, total As Decimal) _
                    From order In ctx.SalesOrderHeaders _
                    Where (order.TotalDue >= total) _
                    Select order)

Sub CompiledQuery2()
    Using context As New AdventureWorksEntities()

        Dim totalDue As Decimal = 200.0

        Dim orders As IQueryable(Of SalesOrderHeader) = s_compQuery2.Invoke(context, totalDue)

        For Each order In orders
            Console.WriteLine("ID: {0}  Order date: {1} Total due: {2}", _
                                order.SalesOrderID, _
                                order.OrderDate, _
                                order.TotalDue)
        Next
    End Using
End Sub
```

Example 2

The following example compiles and then invokes a query that returns an ObjectQuery<T> instance:

```
static readonly Func<AdventureWorksEntities, ObjectQuery<SalesOrderHeader>> s_compiledQuery1 =
    CompiledQuery.Compile<AdventureWorksEntities, ObjectQuery<SalesOrderHeader>>(
            ctx => ctx.SalesOrderHeaders);

static void CompiledQuery1_MQ()
{

    using (AdventureWorksEntities context = new AdventureWorksEntities())
    {
        IQueryable<SalesOrderHeader> orders = s_compiledQuery1.Invoke(context);

        foreach (SalesOrderHeader order in orders)
            Console.WriteLine(order.SalesOrderID);
    }
}
```

```
ReadOnly s_compQuery1 As Func(Of AdventureWorksEntities, ObjectQuery(Of SalesOrderHeader)) = _
    CompiledQuery.Compile(Of AdventureWorksEntities, ObjectQuery(Of SalesOrderHeader))( _
                Function(ctx) ctx.SalesOrderHeaders)

Sub CompiledQuery1_MQ()

    Using context As New AdventureWorksEntities()

        Dim orders As ObjectQuery(Of SalesOrderHeader) = s_compQuery1.Invoke(context)

        For Each order In orders
            Console.WriteLine(order.SalesOrderID)
        Next

    End Using
End Sub
```

Example 3

The following example compiles and then invokes a query that returns the average of the product list prices as a Decimal value:

```
static readonly Func<AdventureWorksEntities, Decimal> s_compiledQuery3MQ =
CompiledQuery.Compile<AdventureWorksEntities, Decimal>(
            ctx => ctx.Products.Average(product => product.ListPrice));

static void CompiledQuery3_MQ()
{

    using (AdventureWorksEntities context = new AdventureWorksEntities())
    {
        Decimal averageProductPrice = s_compiledQuery3MQ.Invoke(context);

        Console.WriteLine("The average of the product list prices is $: {0}", averageProductPrice);
    }
}
```

```
Using context As New AdventureWorksEntities()
    Dim compQuery = CompiledQuery.Compile(Of AdventureWorksEntities, Decimal)( _
            Function(ctx) ctx.Products.Average(Function(Product) Product.ListPrice))

    Dim averageProductPrice As Decimal = compQuery.Invoke(context)

    Console.WriteLine("The average of the product list prices is $: {0}", averageProductPrice)
End Using
```

Example 4

The following example compiles and then invokes a query that accepts a String input parameter and then returns a `Contact` whose email address starts with the specified string:

```
static readonly Func<AdventureWorksEntities, string, Contact> s_compiledQuery4MQ =
    CompiledQuery.Compile<AdventureWorksEntities, string, Contact>(
            (ctx, name) => ctx.Contacts.First(contact => contact.EmailAddress.StartsWith(name)));

static void CompiledQuery4_MQ()
{
    using (AdventureWorksEntities context = new AdventureWorksEntities())
    {
        string contactName = "caroline";
        Contact foundContact = s_compiledQuery4MQ.Invoke(context, contactName);

        Console.WriteLine("An email address starting with 'caroline': {0}",
            foundContact.EmailAddress);
    }
}
```

```
Using context As New AdventureWorksEntities()
    Dim compQuery = CompiledQuery.Compile(Of AdventureWorksEntities, String, Contact)( _
            Function(ctx, name) ctx.Contacts.First(Function(contact) contact.EmailAddress.StartsWith(name)))

    Dim contactName As String = "caroline"
    Dim foundContact As Contact = compQuery.Invoke(context, contactName)

    Console.WriteLine("An email address starting with 'caroline': {0}", _
            foundContact.EmailAddress)
End Using
```

Example 5

The following example compiles and then invokes a query that accepts DateTime and Decimal input parameters and returns a sequence of orders where the order date is later than March 8, 2003, and the total due is less than $300.00:

```
static readonly Func<AdventureWorksEntities, DateTime, Decimal, IQueryable<SalesOrderHeader>>
s_compiledQuery5 =
    CompiledQuery.Compile<AdventureWorksEntities, DateTime, Decimal, IQueryable<SalesOrderHeader>>(
            (ctx, orderDate, totalDue) => from product in ctx.SalesOrderHeaders
                                          where product.OrderDate > orderDate
                                             && product.TotalDue < totalDue
                                          orderby product.OrderDate
                                          select product);

static void CompiledQuery5()
{
    using (AdventureWorksEntities context = new AdventureWorksEntities())
    {
        DateTime date = new DateTime(2003, 3, 8);
        Decimal amountDue = 300.00M;

        IQueryable<SalesOrderHeader> orders = s_compiledQuery5.Invoke(context, date, amountDue);

        foreach (SalesOrderHeader order in orders)
        {
            Console.WriteLine("ID: {0} Order date: {1} Total due: {2}", order.SalesOrderID, order.OrderDate,
order.TotalDue);
        }
    }
}
```

```
ReadOnly s_compQuery5 = _
   CompiledQuery.Compile(Of AdventureWorksEntities, DateTime, Decimal, IQueryable(Of SalesOrderHeader))( _
               Function(ctx, orderDate, totalDue) From product In ctx.SalesOrderHeaders _
                                                  Where product.OrderDate > orderDate _
                                                     And product.TotalDue < totalDue _
                                                  Order By product.OrderDate _
                                                  Select product)
Sub CompiledQuery5()

    Using context As New AdventureWorksEntities()

        Dim orderedAfterDate As DateTime = New DateTime(2003, 3, 8)
        Dim amountDue As Decimal = 300.0

        Dim orders As IQueryable(Of SalesOrderHeader) = _
            s_compQuery5.Invoke(context, orderedAfterDate, amountDue)

        For Each order In orders
            Console.WriteLine("ID: {0} Order date: {1} Total due: {2}", _
                              order.SalesOrderID, order.OrderDate, order.TotalDue)
        Next

    End Using
End Sub
```

Example 6

The following example compiles and then invokes a query that accepts a DateTime input parameter and returns a sequence of orders where the order date is later than March 8, 2004. This query returns the order information as a sequence of anonymous types. Anonymous types are inferred by the compiler, so you cannot specify type parameters in the CompiledQuery's `Compile` method and the type is defined in the query itself.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var compiledQuery = CompiledQuery.Compile((AdventureWorksEntities ctx, DateTime orderDate) =>
        from order in ctx.SalesOrderHeaders
        where order.OrderDate > orderDate
        select new {order.OrderDate, order.SalesOrderID, order.TotalDue});

    DateTime date = new DateTime(2004, 3, 8);
    var results = compiledQuery.Invoke(context, date);

    foreach (var order in results)
    {
        Console.WriteLine("ID: {0} Order date: {1} Total due: {2}", order.SalesOrderID, order.OrderDate,
order.TotalDue);
    }
}
```

```
Using context As New AdventureWorksEntities()
    Dim compQuery = CompiledQuery.Compile( _
            Function(ctx As AdventureWorksEntities, orderDate As DateTime) _
                From order In ctx.SalesOrderHeaders _
                Where order.OrderDate > orderDate _
                Select New With {order.OrderDate, order.SalesOrderID, order.TotalDue})

    Dim orderedAfterDate As DateTime = New DateTime(2004, 3, 8)

    Dim orders = compQuery.Invoke(context, orderedAfterDate)

    For Each order In orders
        Console.WriteLine("ID: {0} Order date: {1} Total due: {2}", _
                          order.SalesOrderID, order.OrderDate, order.TotalDue)
    Next

End Using
```

Example 7

The following example compiles and then invokes a query that accepts a user-defined structure input parameter and returns a sequence of orders. The structure defines start date, end date, and total due query parameters, and the query returns orders shipped between March 3 and March 8, 2003 with a total due greater than $700.00.

```
static Func<AdventureWorksEntities, MyParams, IQueryable<SalesOrderHeader>> s_compiledQuery =
    CompiledQuery.Compile<AdventureWorksEntities, MyParams, IQueryable<SalesOrderHeader>>(
            (ctx, myparams) => from sale in ctx.SalesOrderHeaders
                               where sale.ShipDate > myparams.startDate && sale.ShipDate < myparams.endDate
                               && sale.TotalDue > myparams.totalDue
                               select sale);
static void CompiledQuery7()
{

    using (AdventureWorksEntities context = new AdventureWorksEntities())
    {
        MyParams myParams = new MyParams();
        myParams.startDate = new DateTime(2003, 3, 3);
        myParams.endDate = new DateTime(2003, 3, 8);
        myParams.totalDue = 700.00M;

        IQueryable<SalesOrderHeader> sales = s_compiledQuery.Invoke(context, myParams);

        foreach (SalesOrderHeader sale in sales)
        {
            Console.WriteLine("ID: {0}", sale.SalesOrderID);
            Console.WriteLine("Ship date: {0}", sale.ShipDate);
            Console.WriteLine("Total due: {0}", sale.TotalDue);
        }
    }
}
```

```
ReadOnly s_compQuery = CompiledQuery.Compile(Of AdventureWorksEntities, MyParams, IQueryable(Of
SalesOrderHeader))( _
                Function(ctx, mySearchParams) _
                    From sale In ctx.SalesOrderHeaders _
                    Where sale.ShipDate > mySearchParams.startDate _
                       And sale.ShipDate < mySearchParams.endDate _
                       And sale.TotalDue > mySearchParams.totalDue _
                    Select sale)

Sub CompiledQuery7()

    Using context As New AdventureWorksEntities()

        Dim myParams As MyParams = New MyParams()
        myParams.startDate = New DateTime(2003, 3, 3)
        myParams.endDate = New DateTime(2003, 3, 8)
        myParams.totalDue = 700.0

        Dim sales = s_compQuery.Invoke(context, myParams)

        For Each sale In sales
            Console.WriteLine("ID: {0}", sale.SalesOrderID)
            Console.WriteLine("Ship date: {0}", sale.ShipDate)
            Console.WriteLine("Total due: {0}", sale.TotalDue)
        Next

    End Using
End Sub
```

The structure that defines the query parameters:

```
struct MyParams
{
    public DateTime startDate;
    public DateTime endDate;
    public decimal totalDue;
}
```

```
Public Structure MyParams
    Public startDate As DateTime
    Public endDate As DateTime
    Public totalDue As Decimal
End Structure
```

See also

- ADO.NET Entity Framework
- LINQ to Entities
- EF Merge Options and Compiled Queries

Query Execution

11/8/2022 • 9 minutes to read • Edit Online

After a LINQ query is created by a user, it is converted to a command tree. A command tree is a representation of a query that is compatible with the Entity Framework. The command tree is then executed against the data source. At query execution time, all query expressions (that is, all components of the query) are evaluated, including those expressions that are used in result materialization.

At what point query expressions are executed can vary. LINQ queries are always executed when the query variable is iterated over, not when the query variable is created. This is called *deferred execution*. You can also force a query to execute immediately, which is useful for caching query results. This is described later in this topic.

When a LINQ to Entities query is executed, some expressions in the query might be executed on the server and some parts might be executed locally on the client. Client-side evaluation of an expression takes place before the query is executed on the server. If an expression is evaluated on the client, the result of that evaluation is substituted for the expression in the query, and the query is then executed on the server. Because queries are executed on the data source, the data source configuration overrides the behavior specified in the client. For example, null value handling and numerical precision depend on the server settings. Any exceptions thrown during query execution on the server are passed directly up to the client.

> **TIP**
> For a convenient summary of query operators in table format, which lets you quickly identify an operator's execution behavior, see Classification of Standard Query Operators by Manner of Execution (C#).

Deferred query execution

In a query that returns a sequence of values, the query variable itself never holds the query results and only stores the query commands. Execution of the query is deferred until the query variable is iterated over in a `foreach` or `For Each` loop. This is known as *deferred execution*; that is, query execution occurs some time after the query is constructed. This means that you can execute a query as frequently as you want to. This is useful when, for example, you have a database that is being updated by other applications. In your application, you can create a query to retrieve the latest information and repeatedly execute the query, returning the updated information every time.

Deferred execution enables multiple queries to be combined or a query to be extended. When a query is extended, it is modified to include the new operations, and the eventual execution will reflect the changes. In the following example, the first query returns all the products. The second query extends the first by using `Where` to return all the products of size "L":

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<Product> productsQuery =
        from p in context.Products
        select p;

    IQueryable<Product> largeProducts = productsQuery.Where(p => p.Size == "L");

    Console.WriteLine("Products of size 'L':");
    foreach (var product in largeProducts)
    {
        Console.WriteLine(product.Name);
    }
}
```

```
Using context As New AdventureWorksEntities()
    Dim productsQuery = _
        From p In context.Products _
        Select p

    Dim largeProducts = _
        productsQuery.Where(Function(p) p.Size = "L")

    Console.WriteLine("Products of size 'L':")
    For Each product In largeProducts
        Console.WriteLine(product.Name)
    Next
End Using
```

After a query has been executed all successive queries will use the in-memory LINQ operators. Iterating over the query variable by using a `foreach` or `For Each` statement or by calling one of the LINQ conversion operators will cause immediate execution. These conversion operators include the following: ToList, ToArray, ToLookup, and ToDictionary.

Immediate Query Execution

In contrast to the deferred execution of queries that produce a sequence of values, queries that return a singleton value are executed immediately. Some examples of singleton queries are Average, Count, First, and Max. These execute immediately because the query must produce a sequence to calculate the singleton result. You can also force immediate execution. This is useful when you want to cache the results of a query. To force immediate execution of a query that does not produce a singleton value, you can call the ToList method, the ToDictionary method, or the ToArray method on a query or query variable. The following example uses the ToArray method to immediately evaluate a sequence into an array.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    ObjectSet<Product> products = context.Products;

    Product[] prodArray = (
        from product in products
        orderby product.ListPrice descending
        select product).ToArray();

    Console.WriteLine("Every price from highest to lowest:");
    foreach (Product product in prodArray)
    {
        Console.WriteLine(product.ListPrice);
    }
}
```

```
Using context As New AdventureWorksEntities
    Dim products As ObjectSet(Of Product) = context.Products

    Dim prodArray As Product() = ( _
        From product In products _
        Order By product.ListPrice Descending _
        Select product).ToArray()

    Console.WriteLine("The list price from highest to lowest:")
    For Each prod As Product In prodArray
        Console.WriteLine(prod.ListPrice)
    Next
End Using
```

You could also force execution by putting the `foreach` or `For Each` loop immediately after the query expression, but by calling ToList or ToArray you cache all the data in a single collection object.

Store Execution

In general, expressions in LINQ to Entities are evaluated on the server, and the behavior of the expression should not be expected to follow common language runtime (CLR) semantics, but those of the data source. There are exceptions to this, however, such as when the expression is executed on the client. This could cause unexpected results, for example when the server and client are in different time zones.

Some expressions in the query might be executed on the client. In general, most query execution is expected to occur on the server. Aside from methods executed against query elements mapped to the data source, there are often expressions in the query that can be executed locally. Local execution of a query expression yields a value that can be used in the query execution or result construction.

Certain operations are always executed on the client, such as binding of values, sub expressions, sub queries from closures, and materialization of objects into query results. The net effect of this is that these elements (for example, parameter values) cannot be updated during the execution. Anonymous types can be constructed inline on the data source, but should not be assumed to do so. Inline groupings can be constructed in the data source, as well, but this should not be assumed in every instance. In general, it is best not to make any assumptions about what is constructed on the server.

This section describes the scenarios in which code is executed locally on the client. For more information about which types of expressions are executed locally, see Expressions in LINQ to Entities Queries.

Literals and Parameters

Local variables, such as the `orderID` variable in the following example, are evaluated on the client.

```
int orderID = 51987;

IQueryable<SalesOrderHeader> salesInfo =
    from s in context.SalesOrderHeaders
    where s.SalesOrderID == orderID
    select s;
```

```
Dim orderID As Integer = 51987

Dim salesInfo = _
    From s In context.SalesOrderHeaders _
    Where s.SalesOrderID = orderID _
    Select s
```

Method parameters are also evaluated on the client. The `orderID` parameter passed into the `MethodParameterExample` method, below, is an example.

```
public static void MethodParameterExample(int orderID)
{
    using (AdventureWorksEntities context = new AdventureWorksEntities())
    {

        IQueryable<SalesOrderHeader> salesInfo =
            from s in context.SalesOrderHeaders
            where s.SalesOrderID == orderID
            select s;

        foreach (SalesOrderHeader sale in salesInfo)
        {
            Console.WriteLine("OrderID: {0}, Total due: {1}", sale.SalesOrderID, sale.TotalDue);
        }
    }
}
```

```
Function MethodParameterExample(ByVal orderID As Integer)
    Using context As New AdventureWorksEntities()

        Dim salesInfo = _
            From s In context.SalesOrderHeaders _
            Where s.SalesOrderID = orderID _
            Select s

        Console.WriteLine("Sales order info:")
        For Each sale As SalesOrderHeader In salesInfo
            Console.WriteLine("OrderID: {0}, Total due: {1}", sale.SalesOrderID, sale.TotalDue)
        Next
    End Using

End Function
```

Casting Literals on the Client

Casting from `null` to a CLR type is executed on the client:

```
IQueryable<Contact> query =
    from c in context.Contacts
    where c.EmailAddress == (string)null
    select c;
```

```
Dim query = _
    From c In context.Contacts _
    Where c.EmailAddress = CType(Nothing, String) _
    Select c
```

Casting to a type, such as a nullable Decimal, is executed on the client:

```
var weight = (decimal?)23.77;
IQueryable<Product> query =
    from product in context.Products
    where product.Weight == weight
    select product;
```

```
Dim weight = CType(23.77, Decimal?)
Dim query = _
    From product In context.Products _
    Where product.Weight = weight _
    Select product
```

Constructors for Literals

New CLR types that can be mapped to conceptual model types are executed on the client:

```
var weight = new decimal(23.77);
IQueryable<Product> query =
    from product in context.Products
    where product.Weight == weight
    select product;
```

```
Dim weight = New Decimal(23.77)
Dim query = _
    From product In context.Products _
    Where product.Weight = weight _
    Select product
```

New arrays are also executed on the client.

Store Exceptions

Any store errors that are encountered during query execution are passed up to the client, and are not mapped or handled.

Store Configuration

When the query executes on the store, the store configuration overrides all client behaviors, and store semantics are expressed for all operations and expressions. This can result in a difference in behavior between CLR and store execution in areas such as null comparisons, GUID ordering, precision and accuracy of operations involving non-precise data types (such as floating point types or DateTime), and string operations. It is important to keep this in mind when examining query results.

For example, the following are some differences in behavior between the CLR and SQL Server:

- SQL Server orders GUIDs differently than the CLR.
- There can also be differences in result precision when dealing with the Decimal type on SQL Server. This is due to the fixed precision requirements of the SQL Server decimal type. For example, the average of Decimal values 0.0, 0.0, and 1.0 is 0.33333333333333333333333333333 in memory on the client, but 0.333333 in the store (based on the default precision for SQL Server's decimal type).
- Some string comparison operations are also handled differently in SQL Server than in the CLR. String comparison behavior depends on the collation settings on the server.
- Function or method calls, when included in a LINQ to Entities query, are mapped to canonical functions in the Entity Framework, which are then translated to Transact-SQL and executed on the SQL Server database. There are cases when the behavior these mapped functions exhibit might differ from the implementation in the base class libraries. For example, calling the Contains, StartsWith, and EndsWith methods with an empty string as a parameter will return `true` when executed in the CLR, but will return `false` when executed in SQL Server. The EndsWith method can also return different results because SQL Server considers two strings to be equal if they only differ in trailing white space, whereas the CLR

considers them to be not equal. This is illustrated by the following example:

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    IQueryable<string> query = from p in context.Products
                               where p.Name == "Reflector"
                               select p.Name;

    IEnumerable<bool> q = query.Select(c => c.EndsWith("Reflector "));

    Console.WriteLine("LINQ to Entities returns: " + q.First());
    Console.WriteLine("CLR returns: " + "Reflector".EndsWith("Reflector "));
}
```

```
Using context As New AdventureWorksEntities()

    Dim query = _
        From p In context.Products _
        Where p.Name = "Reflector" _
        Select p.Name

    Dim q = _
        query.Select(Function(c) c.EndsWith("Reflector "))

    Console.WriteLine("LINQ to Entities returns: " & q.First())
    Console.WriteLine("CLR returns: " & "Reflector".EndsWith("Reflector "))
End Using
```

Query Results

11/8/2022 • 3 minutes to read • Edit Online

After a LINQ to Entities query is converted to command trees and executed, the query results are usually returned as one of the following:

- A collection of zero or more typed entity objects or a projection of complex types in the conceptual model.
- CLR types supported by the conceptual model.
- Inline collections.
- Anonymous types.

When the query has executed against the data source, the results are materialized into CLR types and returned to the client. All object materialization is performed by the Entity Framework. Any errors that result from an inability to map between the Entity Framework and the CLR will cause exceptions to be thrown during object materialization.

If the query execution returns primitive conceptual model types, the results consist of CLR types that are stand-alone and disconnected from the Entity Framework. However, if the query returns a collection of typed entity objects, represented by ObjectQuery<T>, those types are tracked by the object context. All object behavior (such as child/parent collections, change tracking, polymorphism, and so on) are as defined in the Entity Framework. This functionality can be used in its capacity, as defined in the Entity Framework. For more information, see Working with Objects.

Struct types returned from queries (such as anonymous types and nullable complex types) can be of `null` value. An EntityCollection<TEntity> property of a returned entity can also be of `null` value. This can result from projecting the collection property of an entity that is of `null` value, such as calling FirstOrDefault on an ObjectQuery<T> that has no elements.

In certain situations, a query might appear to generate a materialized result during its execution, but the query will be executed on the server and the entity object will never be materialized in the CLR. This can cause problems if you are depending on side effects of object materialization.

The following example contains a custom class, `MyContact`, with a `LastName` property. When the `LastName` property is set, a `count` variable is incremented. If you execute the two following queries, the first query will increment `count` while the second query will not. This is because in the second query the `LastName` property is projected from the results and the `MyContact` class is never created, because it is not required to execute the query on the store.

```
public static int count = 0;

static void Main(string[] args)
{
    using (AdventureWorksEntities AWEntities = new AdventureWorksEntities())
    {

        var query1 = AWEntities
           .Contacts
           .Where(c => c.LastName == "Jones")
           .Select(c => new MyContact { LastName = c.LastName });

        // Execute the first query and print the count.
        query1.ToList();
        Console.WriteLine("Count: " + count);

        //Reset the count variable.
        count = 0;

        var query2 = AWEntities
           .Contacts
           .Where(c => c.LastName == "Jones")
           .Select(c => new MyContact { LastName = c.LastName })
           .Select(my => my.LastName);

        // Execute the second query and print the count.
        query2.ToList();
        Console.WriteLine("Count: " + count);
    }

    Console.WriteLine("Hit enter...");
    Console.Read();
}
```

```
Public count As Integer = 0

Sub Main()

    Using AWEntities As New AdventureWorksEntities()

        Dim query1 = AWEntities.Contacts _
        .Where(Function(c) c.LastName = "Jones") _
        .Select(Function(c) New MyContact With {.LastName = c.LastName})

        ' Execute the first query and print the count.
        query1.ToList()
        Console.WriteLine("Count: " & count)

        ' Reset the count variable.
        count = 0

        Dim query2 = AWEntities _
        .Contacts() _
        .Where(Function(c) c.LastName = "Jones") _
        .Select(Function(c) New MyContact With {.LastName = c.LastName}) _
        .Select(Function(x) x.LastName)

        ' Execute the second query and print the count.
        query2.ToList()
        Console.WriteLine("Count: " & count)

    End Using
End Sub
```

```
public class MyContact
{

    String _lastName;

    public string LastName
    {
        get
        {
            return _lastName;
        }

        set
        {
            _lastName = value;
            count++;
        }
    }
}
```

```
Public Class MyContact

    Private _lastName As String

    Public Property LastName() As String
        Get
            Return _lastName
        End Get

        Set(ByVal value As String)
            _lastName = value
            count += 1
        End Set
    End Property

End Class
```

Standard Query Operators in LINQ to Entities Queries

11/8/2022 • 6 minutes to read • Edit Online

In a query, you specify the information that you want to retrieve from the data source. A query can also specify how that information should be sorted, grouped, and shaped before it is returned. LINQ provides a set of standard query methods that you can use in a query. Most of these methods operate on sequences; in this context, a sequence is an object whose type implements the IEnumerable<T> interface or the IQueryable<T> interface. The standard query operators query functionality includes filtering, projection, aggregation, sorting, grouping, paging, and more. Some of the more frequently used standard query operators have dedicated keyword syntax so that they can be called by using query expression syntax. A query expression is a different, more readable way to express a query than the method-based equivalent. Query expression clauses are translated into calls to the query methods at compile time. For a list of standard query operators that have equivalent query expression clauses, see Standard Query Operators Overview.

Not all of the standard query operators are supported in LINQ to Entities queries. For more information, see Supported and Unsupported LINQ Methods (LINQ to Entities). This topic provides information about the standard query operators that is specific to LINQ to Entities. For more information about known issues in LINQ to Entities queries, see Known Issues and Considerations in LINQ to Entities.

Projection and Filtering Methods

Projection refers to transforming the elements of a result set into a desired form. For example, you can project a subset of the properties you need from each object in the result set, you can project a property and perform a mathematical calculation on it, or you can project the entire object from the result set. The projection methods are `Select` and `SelectMany`.

Filtering refers to the operation of restricting the result set to contain only those elements that match a specified condition. The filtering method is `Where`.

Most overloads of the projection and filtering methods are supported in LINQ to Entities, with the exception of those that accept a positional argument.

Join Methods

Joining is an important operation in queries that target data sources that have no navigable relationships to each other. A join of two data sources is the association of objects in one data source with objects in the other data source that share a common attribute or property. The join methods are `Join` and `GroupJoin`.

Most overloads of the join methods are supported, with the exception of those that use a IEqualityComparer<T>. This is because the comparer cannot be translated to the data source.

Set Methods

Set operations in LINQ are query operations that base their result sets on the presence or absence of equivalent elements within the same or in another collection (or set). The set methods are `All`, `Any`, `Concat`, `Contains`, `DefaultIfEmpty`, `Distinct`, `EqualAll`, `Except`, `Intersect`, and `Union`.

Most overloads of the set methods are supported in LINQ to Entities, though there are some differences in behavior compared to LINQ to Objects. However, set methods that use an IEqualityComparer<T> are not

supported because the comparer cannot be translated to the data source.

Ordering Methods

Ordering, or sorting, refers to the ordering the elements of a result set based on one or more attributes. By specifying more than one sort criterion, you can break ties within a group.

Most overloads of the ordering methods are supported, with the exception of those that use an IComparer<T>. This is because the comparer cannot be translated to the data source. The ordering methods are `OrderBy`, `OrderByDescending`, `ThenBy`, `ThenByDescending`, and `Reverse`.

Because the query is executed on the data source, the ordering behavior may differ from queries executed in the CLR. This is because ordering options, such as case ordering, kanji ordering, and null ordering, can be set in the data source. Depending on the data source, these ordering options might produce different results than in the CLR.

If you specify the same key selector in more than one ordering operation, a duplicate ordering will be produced. This is not valid and an exception will be thrown.

Grouping Methods

Grouping refers to placing data into groups so that the elements in each group share a common attribute. The grouping method is `GroupBy`.

Most overloads of the grouping methods are supported, with the exception of those that use an IEqualityComparer<T>. This is because the comparer cannot be translated to the data source.

The grouping methods are mapped to the data source using a distinct sub-query for the key selector. The key selector comparison sub-query is executed by using the semantics of the data source, including issues related to comparing `null` values.

Aggregate Methods

An aggregation operation computes a single value from a collection of values. For example, calculating the average daily temperature from a month's worth of daily temperature values is an aggregation operation. The aggregate methods are `Aggregate`, `Average`, `Count`, `LongCount`, `Max`, `Min`, and `Sum`.

Most overloads of the aggregate methods are supported. For behavior related to null values, the aggregate methods use the data source semantics. The behavior of the aggregation methods when null values are involved might be different, depending on which back-end data source is being used. Aggregate method behavior using the semantics of the data source might also be different from what is expected from CLR methods. For example, the default behavior for the `Sum` method on SQL Server is to ignore any null values instead of throwing an exception.

Any exceptions that result from aggregation, such as an overflow from the `Sum` function, are thrown as data source exceptions or Entity Framework exceptions during the materialization of the query results.

For those methods that involve a calculation over a sequence, such as `Sum` or `Average`, the actual calculation is performed on the server. As a result, type conversions and loss of precision might occur on the server, and the results might differ from what is expected using CLR semantics.

The default behavior of the aggregate methods for null/non-null values is shown in the following table:

METHOD	NO DATA	ALL NULL VALUES	SOME NULL VALUES	NO NULL VALUES

METHOD	NO DATA	ALL NULL VALUES	SOME NULL VALUES	NO NULL VALUES
`Average`	Returns null.	Returns null.	Returns the average of the non-null values in a sequence.	Computes the average of a sequence of numeric values.
`Count`	Returns 0	Returns the number of null values in the sequence.	Returns the number of null and non-null values in the sequence.	Returns the number of elements in the sequence.
`Max`	Returns null.	Returns null.	Returns the maximum non-null value in a sequence.	Returns the maximum value in a sequence.
`Min`	Returns null.	Returns null.	Returns the minimum non-null value in a sequence.	Returns the minimum value in a sequence.
`Sum`	Returns null.	Returns null.	Returns the sum of the non-null value in a sequence.	Computes the sum of a sequence of numeric values.

Type Methods

The two LINQ methods that deal with type conversion and testing are both supported in the context of the Entity Framework. This means that the only supported types are types that map to the appropriate Entity Framework type. For a list of these types, see Conceptual Model Types (CSDL). The type methods are `Convert` and `OfType`.

`OfType` is supported for entity types. `Convert` is supported for conceptual model primitive types. The C# `is` and `as` methods are also supported.

Paging Methods

Paging operations return a single element or multiple elements from a sequence. The supported paging methods are `First`, `FirstOrDefault`, `Single`, `SingleOrDefault`, `Skip`, and `Take`.

A number of paging methods are not supported, due either to the inability to map functions to the data source or to the lack of implicit ordering of sets on the data source. Methods that return a default value are restricted to conceptual model primitive types and reference types with null defaults. Paging methods that are executed on an empty sequence will return null.

See also

- Supported and Unsupported LINQ Methods (LINQ to Entities)
- Standard Query Operators Overview

CLR Method to Canonical Function Mapping

11/8/2022 • 6 minutes to read • Edit Online

The Entity Framework provides a set of canonical functions that implement functionality that is common across many database systems, such as string manipulation and mathematical functions. This enables developers to target a broad range of database systems. When called from a querying technology, such as LINQ to Entities, these canonical functions are translated to the correct corresponding store function for the provider being used. This allows function invocations to be expressed in a common form across data sources, providing a consistent query experience across data sources. The bitwise AND, OR, NOT, and XOR operators are also mapped to canonical functions when the operand is a numeric type. For Boolean operands, the bitwise AND, OR, NOT, and XOR operators compute the logical AND, OR, NOT, and XOR operations of their operands. For more information, see Canonical Functions.

For LINQ scenarios, queries against the Entity Framework involve mapping certain CLR methods to methods on the underlying data source through canonical functions. Any method calls in a LINQ to Entities query that are not explicitly mapped to a canonical function will result in a runtime NotSupportedException exception being thrown.

System.String Method (Static) Mapping

SYSTEM.STRING METHOD (STATIC)	CANONICAL FUNCTION
System.String Concat(String `str0`, String `str1`)	Concat(`str0` , `str1`)
System.String Concat(String `str0`, String `str1`, String `str2`)	Concat(Concat(`str0` , `str1`), `str2`)
System.String Concat(String `str0`, String `str1`, String `str2`, String `str03`)	Concat(Concat(Concat(`str0` , `str1`), `str2`), `str3`)
Boolean Equals(String `a`, String `b`)	= operator
Boolean IsNullOrEmpty(String `value`)	(IsNull(`value`)) OR Length(`value`) = 0
Boolean op_Equality(String `a`, String `b`)	= operator
Boolean op_Inequality(String `a`, String `b`)	!= operator
Microsoft.VisualBasic.Strings.Trim(String `str`)	Trim(`str`)
Microsoft.VisualBasic.Strings.LTrim(String `str`)	Ltrim(`str`)
Microsoft.VisualBasic.Strings.RTrim(String `str`)	Rtrim(`str`)
Microsoft.VisualBasic.Strings.Len(String `expression`)	Length(`expression`)
Microsoft.VisualBasic.Strings.Left(String `str`, Int32 `Length`)	Left(`str` , `Length`)

SYSTEM.STRING METHOD (STATIC)	CANONICAL FUNCTION
Microsoft.VisualBasic.Strings.Mid(String `str`, Int32 `Start`, Int32 `Length`)	Substring(`str`, `Start`, `Length`)
Microsoft.VisualBasic.Strings.Right(String `str`, Int32 `Length`)	Right(`str`, `Length`)
Microsoft.VisualBasic.Strings.UCase(String `Value`)	ToUpper(`Value`)
Microsoft.VisualBasic.Strings.LCase(String Value)	ToLower(`Value`)

System.String Method (Instance) Mapping

SYSTEM.STRING METHOD (INSTANCE)	CANONICAL FUNCTION	NOTES
Boolean Contains(String `value`)	`this` LIKE '% `value` %'	If `value` is not a constant, then this maps to IndexOf(`this`, `value`) > 0
Boolean EndsWith(String `value`)	`this` LIKE `'` % `value` '	If `value` is not a constant, then this maps to Right(`this`, length(`value`)) = `value`.
Boolean StartsWith(String `value`)	`this` LIKE ' `value` %'	If `value` is not a constant, then this maps to IndexOf(`this`, `value`) = 1.
Length	Length(`this`)	
Int32 IndexOf(String `value`)	IndexOf(`this`, `value`) - 1	
System.String Insert(Int32 `startIndex`, String `value`)	Concat(Concat(Substring(`this`, 1, `startIndex`), `value`), Substring(`this`, `startIndex` +1, Length(`this`) - `startIndex`))	
System.String Remove(Int32 `startIndex`)	Substring(`this`, 1, `startIndex`)	
System.String Remove(Int32 `startIndex`, Int32 `count`)	Concat(Substring(`this`, 1, `startIndex`), Substring(`this`, `startIndex` + `count` +1, Length(`this`) - (`startIndex` + `count`)))	Remove(`startIndex`, `count`) is only supported if `count` is an integer greater than or equal to 0.
System.String Replace(String `oldValue`, String `newValue`)	Replace(`this`, `oldValue`, `newValue`)	
System.String Substring(Int32 `startIndex`)	Substring(`this`, `startIndex` +1, Length(`this`) - `startIndex`)	
System.String Substring(Int32 `startIndex`, Int32 `length`)	Substring(`this`, `startIndex` +1, `length`)	

SYSTEM.STRING METHOD (INSTANCE)	CANONICAL FUNCTION	NOTES
System.String ToLower()	ToLower(`this`)	
System.String ToUpper()	ToUpper(`this`)	
System.String Trim()	Trim(`this`)	
System.String TrimEnd(Char[] `trimChars`)	RTrim(`this`)	
System.String TrimStart(Char[] `trimChars`)	LTrim(`this`)	
Boolean Equals(String `value`)	= operator	

System.DateTime Method (Static) Mapping

SYSTEM.DATETIME METHOD (STATIC)	CANONICAL FUNCTION	NOTES
Boolean Equals(DateTime `t1` , DateTime `t2`)	= operator	
System.DateTime.Now	CurrentDateTime()	
System.DateTime.UtcNow	CurrentUtcDateTime()	
Boolean op_Equality(DateTime `d1` , DateTime `d2`)	= operator	
Boolean op_GreaterThan(DateTime `t1` , DateTime `t2`)	> operator	
Boolean op_GreaterThanOrEqual(DateTime `t1` , DateTime `t2`)	>= operator	
Boolean op_Inequality(DateTime `t1` , DateTime `t2`)	!= operator	
Boolean op_LessThan(DateTime `t1` , DateTime `t2`)	< operator	
Boolean op_LessThanOrEqual(DateTime `t1` , DateTime `t2`)	<= operator	

SYSTEM.DATETIME METHOD (STATIC)	CANONICAL FUNCTION	NOTES
Microsoft.VisualBasic.DateAndTime.DatePart(_ ByVal `Interval` As DateInterval, _ ByVal `DateValue` As DateTime, _ Optional ByVal `FirstDayOfWeekValue` As FirstDayOfWeek = VbSunday, _ Optional ByVal `FirstWeekOfYearValue` As FirstWeekOfYear = VbFirstJan1 _) As Integer		See the DatePart Function section for more information.
Microsoft.VisualBasic.DateAndTime.Now	CurrentDateTime()	
Microsoft.VisualBasic.DateAndTime.Year(DateTime `TimeValue`)	Year()	
Microsoft.VisualBasic.DateAndTime.Month(DateTime `TimeValue`)	Month()	
Microsoft.VisualBasic.DateAndTime.Day(DateTime `TimeValue`)	Day()	
Microsoft.VisualBasic.DateAndTime.Hour(DateTime `TimeValue`)	Hour()	
Microsoft.VisualBasic.DateAndTime.Minute(DateTime `TimeValue`)	Minute()	
Microsoft.VisualBasic.DateAndTime.Second(DateTime `TimeValue`)	Second()	

System.DateTime Method (Instance) Mapping

SYSTEM.DATETIME METHOD (INSTANCE)	CANONICAL FUNCTION
Boolean Equals(DateTime `value`)	= operator
Day	Day(`this`)
Hour	Hour(`this`)
Millisecond	Millisecond(`this`)
Minute	Minute(`this`)

SYSTEM.DATETIME METHOD (INSTANCE)	CANONICAL FUNCTION
Month	Month(`this`)
Second	Second(`this`)
Year	Year(`this`)

System.DateTimeOffset Method (Instance) Mapping

The mapping shown for the `get` methods on the listed properties.

SYSTEM.DATETIMEOFFSET METHOD (INSTANCE)	CANONICAL FUNCTION	NOTES
Day	Day(`this`)	Not supported against SQL Server 2005.
Hour	Hour(`this`)	Not supported against SQL Server 2005.
Millisecond	Millisecond(`this`)	Not supported against SQL Server 2005.
Minute	Minute(`this`)	Not supported against SQL Server 2005.
Month	Month(`this`)	Not supported against SQL Server 2005.
Second	Second(`this`)	Not supported against SQL Server 2005.
Year	Year(`this`)	Not supported against SQL Server 2005.

NOTE

The Equals method returns `true` if the compared DateTimeOffset objects are equal; `false` otherwise. The CompareTo method returns 0, 1, or -1 depending on whether the compared DateTimeOffset object is equal, greater than, or less than, respectively.

System.DateTimeOffset Method (Static) Mapping

The mapping shown for the `get` methods on the listed properties.

SYSTEM.DATETIMEOFFSET METHOD (STATIC)	CANONICAL FUNCTION	NOTES
System.DateTimeOffset.Now()	CurrentDateTimeOffset()	Not supported against SQL Server 2005.

System.TimeSpan Method (Instance) Mapping

The mapping shown for the `get` methods on the listed properties.

SYSTEM.TIMESPAN METHOD (INSTANCE)	CANONICAL FUNCTION	NOTES
Hours	Hour(`this`)	Not supported against SQL Server 2005.
Milliseconds	Millisecond(`this`)	Not supported against SQL Server 2005.
Minutes	Minute(`this`)	Not supported against SQL Server 2005.
Seconds	Second(`this`)	Not supported against SQL Server 2005.

> **NOTE**
> The Equals method returns `true` if the compared TimeSpan objects are equal; `false` otherwise. The CompareTo method returns 0, 1, or -1 depending on whether the compared TimeSpan object is equal, greater than, or less than, respectively.

DatePart Function

The `DatePart` Function is mapped to one of several different canonical functions, depending on the value of `Interval`. The following table displays the canonical function mapping for the supported values of `Interval`:

INTERVAL VALUE	CANONICAL FUNCTION
DateInterval.Year	Year()
DateInterval.Month	Month()
DateInterval.Day	Day()
DateInterval.Hour	Hour()
DateInterval.Minute	Minute()
DateInterval.Second	Second()

Mathematical Function Mapping

CLR METHOD	CANONICAL FUNCTION
System.Decimal.Ceiling(Decimal `d`)	Ceiling(`d`)
System.Decimal.Floor(Decimal `d`)	Floor(`d`)
System.Decimal.Round(Decimal `d`)	Round(`d`)
System.Math.Ceiling(Decimal `d`)	Ceiling(`d`)

CLR METHOD	CANONICAL FUNCTION
System.Math.Floor(Decimal `d`)	Floor(`d`)
System.Math.Round(Decimal `d`)	Round(`d`)
System.Math.Ceiling(Double `a`)	Ceiling(`a`)
System.Math.Floor(Double `a`)	Floor(`a`)
System.Math.Round(Double `a`)	Round(`a`)
System.Math.Round(Double value, Int16 digits)	Round(value, digits)
System.Math.Round(Double value, Int32 digits)	Round(value, digits)
System.Math.Round(Decimal value, Int16 digits)	Round(value, digits)
System.Math.Round(Decimal value, Int32, digits)	Round(value, digits)
System.Math.Abs(Int16 value)	Abs(value)
System.Math.Abs(Int32 value)	Abs(value)
System.Math.Abs(Int64 value)	Abs(value)
System.Math.Abs(Byte value)	Abs(value)
System.Math.Abs(Single value)	Abs(value)
System.Math.Abs(Double value)	Abs(value)
System.Math.Abs(Decimal value)	Abs(value)
System.Math.Truncate(Double value, Int16 digits)	Truncate(value, digits)
System.Math.Truncate(Double value, Int32 digits)	Truncate(value, digits)
System.Math.Truncate(Decimal value, Int16 digits)	Truncate(value, digits)
System.Math.Truncate(Decimal value, Int32 digits)	Truncate(value, digits)
System.Math.Power(Int32 value, Int64 exponent)	Power(value, exponent)
System.Math.Power(Int32 value, Double exponent)	Power(value, exponent)
System.Math.Power(Int32 value, Decimal exponent)	Power(value, exponent)
System.Math.Power(Int64 value, Int64 exponent)	Power(value, exponent)
System.Math.Power(Int64 value, Double exponent)	Power(value, exponent)

CLR METHOD	CANONICAL FUNCTION
System.Math.Power(Int64 value, Decimal exponent)	Power(value, exponent)
System.Math.Power(Double value, Int64 exponent)	Power(value, exponent)
System.Math.Power(Double value, Double exponent)	Power(value, exponent)
System.Math.Power(Double value, Decimal exponent)	Power(value, exponent)
System.Math.Power(Decimal value, Int64 exponent)	Power(value, exponent)
System.Math.Power(Decimal value, Double exponent)	Power(value, exponent)
System.Math.Power(Decimal value, Decimal exponent)	Power(value, exponent)

Bitwise Operator Mapping

BITWISE OPERATOR	CANONICAL FUNCTION FOR NON-BOOLEAN OPERANDS	CANONICAL FUNCTION FOR BOOLEAN OPERANDS
Bitwise AND operator	BitWiseAnd	op1 AND op2
Bitwise OR operator	BitWiseOr	op1 OR op2
Bitwise NOT operator	BitWiseNot	NOT(op)
Bitwise XOR operator	BitWiseXor	((op1 AND NOT(op2)) OR (NOT(op1) AND op2))

Other Mapping

METHOD	CANONICAL FUNCTION
Guid.NewGuid()	NewGuid()

See also

- LINQ to Entities

Supported and Unsupported LINQ Methods (LINQ to Entities)

11/8/2022 • 20 minutes to read • Edit Online

This section provides information about the Language-Integrated Query (LINQ) standard query operators that are supported or unsupported in LINQ to Entities queries. Many of the LINQ standard query operators have an overloaded version that accepts an integer argument. The integer argument corresponds to a zero-based index in the sequence that is being operated on, an IEqualityComparer<T>, or IComparer<T>. Unless otherwise specified, these overloaded versions of the LINQ standard query operators are not supported, and attempting to use them will throw an exception.

Projection and Restriction Methods

Most of the LINQ projection and restriction methods are supported in LINQ to Entities queries, with the exception of those that accept a positional argument. For more information, see Standard Query Operators in LINQ to Entities Queries. The following table lists the supported and unsupported projection and restriction methods.

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
Select	Supported	`Function Select(Of TSource, TResult) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, TResult)) _ ) As IQueryable(Of TResult)`	`IQueryable<TResult> Select<TSource, TResult> ( this IQueryable<TSource> source, Expression<Func<TSource, TResult>> selector )`
Select	Not supported	`Function Select(Of TSource, TResult) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Integer, TResult)) _ ) As IQueryable(Of TResult)`	`IQueryable<TResult> Select<TSource, TResult> ( this IQueryable<TSource> source, Expression<Func<TSource, int, TResult>> selector )`
SelectMany	Supported	`Function SelectMany(Of TSource, TResult) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, IEnumerable(Of TResult))) _ ) As IQueryable(Of TResult)`	`IQueryable<TResult> SelectMany<TSource, TResult>( this IQueryable<TSource> source, Expression<Func<TSource, IEnumerable<TResult>>> selector )`

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
SelectMany	Not supported	`Function SelectMany(Of TSource, TResult) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Integer, IEnumerable(Of TResult))) _ ) As IQueryable(Of TResult)`	`IQueryable<TResult> SelectMany<TSource, TResult>( this IQueryable<TSource> source, Expression<Func<TSource, int, IEnumerable<TResult>>> selector )`
SelectMany	Supported	`Function SelectMany(Of TSource, TCollection, TResult) ( _ source As IQueryable(Of TSource), _ collectionSelector As Expression(Of Func(Of TSource, IEnumerable(Of TCollection))), _ resultSelector As Expression(Of Func(Of TSource, TCollection, TResult)) _ ) As IQueryable(Of TResult)`	`IQueryable<TResult> SelectMany\<TSource, TCollection, TResult>( this IQueryable<TSource> source, Expression<Func<TSource, IEnumerable<TCollection>>> collectionSelector, Expression<Func\<TSource, TCollection, TResult>> resultSelector )`
SelectMany	Not supported	`Function SelectMany(Of TSource, TCollection, TResult) ( _ source As IQueryable(Of TSource), _ collectionSelector As Expression(Of Func(Of TSource, Integer, IEnumerable(Of TCollection))), _ resultSelector As Expression(Of Func(Of TSource, TCollection, TResult)) _ ) As IQueryable(Of TResult)`	`IQueryable<TResult> SelectMany\<TSource, TCollection, TResult>( this IQueryable<TSource> source, Expression<Func<TSource, int, IEnumerable<TCollection>>> collectionSelector, Expression<Func\<TSource, TCollection, TResult>> resultSelector )`
Where	Supported	`Function Where(Of TSource) ( _ source As IQueryable(Of TSource), _ predicate As Expression(Of Func(Of TSource, Boolean)) _ ) As IQueryable(Of TSource)`	`IQueryable<TSource> Where<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, bool>> predicate )`
Where	Not supported	`Function Where(Of TSource) ( _ source As IQueryable(Of TSource), _ predicate As Expression(Of Func(Of TSource, Integer, Boolean)) _ ) As IQueryable(Of TSource)`	`IQueryable<TSource> Where<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, int, bool>> predicate )`

Join Methods

The LINQ join methods are supported in LINQ to Entities, with the exception of those that accept an `IEqualityComparer` because the comparer cannot be translated to the data source. For more information, see Standard Query Operators in LINQ to Entities Queries. The following table lists the supported and unsupported join methods.

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
GroupJoin	Supported	`Function GroupJoin(Of TOuter, TInner, TKey, TResult) (_ outer As IQueryable(Of TOuter), _ inner As IEnumerable(Of TInner), _ outerKeySelector As Expression(Of Func(Of TOuter, TKey)), _ innerKeySelector As Expression(Of Func(Of TInner, TKey)), _ resultSelector As Expression(Of Func(Of TOuter, IEnumerable(Of TInner), TResult)) _) As IQueryable(Of TResult)`	`IQueryable<TResult> GroupJoin<TOuter, TInner, TKey, TResult>( this IQueryable<TOuter> outer, IEnumerable<TInner> inner, Expression<Func<TOuter, TKey>> outerKeySelector, Expression<Func<TInner, TKey>> innerKeySelector, Expression<Func<TOuter, IEnumerable<TInner>, TResult>> resultSelector )`
GroupJoin	Not Supported	`Function GroupJoin(Of TOuter, TInner, TKey, TResult) (_ outer As IQueryable(Of TOuter), _ inner As IEnumerable(Of TInner), _ outerKeySelector As Expression(Of Func(Of TOuter, TKey)), _ innerKeySelector As Expression(Of Func(Of TInner, TKey)), _ resultSelector As Expression(Of Func(Of TOuter, IEnumerable(Of TInner), TResult)), _ comparer As IEqualityComparer(Of TKey) _) As IQueryable(Of TResult)`	`IQueryable<TResult> GroupJoin\<TOuter, TInner, TKey, TResult>(this IQueryable<TOuter> outer, IEnumerable<TInner> inner, Expression<Func\ <TOuter, TKey>> outerKeySelector, Expression<Func\ <TInner, TKey>> innerKeySelector, Expression<Func<TOuter, IEnumerable<TInner>, TResult>> resultSelector, IEqualityComparer<TKey> comparer)`
Join	Supported	`Function Join(Of TOuter, TInner, TKey, TResult) (_ outer As IQueryable(Of TOuter), _ inner As IEnumerable(Of TInner), _ outerKeySelector As Expression(Of Func(Of TOuter, TKey)), _ innerKeySelector As Expression(Of Func(Of TInner, TKey)), _ resultSelector As Expression(Of Func(Of TOuter, TInner, TResult)) _) As IQueryable(Of TResult)`	`IQueryable<TResult> Join<TOuter, TInner, TKey, TResult>( this IQueryable<TOuter> outer, IEnumerable<TInner> inner, Expression<Func<TOuter, TKey>> outerKeySelector, Expression<Func<TInner, TKey>> innerKeySelector, Expression<Func<TOuter, TInner, TResult>> resultSelector )`

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
Join	Not Supported	`Function Join(Of TOuter, TInner, TKey, TResult) (_ outer As IQueryable(Of TOuter), _ inner As IEnumerable(Of TInner), _ outerKeySelector As Expression(Of Func(Of TOuter, TKey)), _ innerKeySelector As Expression(Of Func(Of TInner, TKey)), _ resultSelector As Expression(Of Func(Of TOuter, TInner, TResult)), _ comparer As IEqualityComparer(Of TKey) _) As IQueryable(Of TResult)`	`IQueryable<TResult> Join\<TOuter, TInner, TKey, TResult>(this IQueryable<TOuter> outer, IEnumerable<TInner> inner, Expression<Func\ <TOuter, TKey>> outerKeySelector, Expression<Func\ <TInner, TKey>> innerKeySelector, Expression<Func\ <TOuter, TInner, TResult>> resultSelector, IEqualityComparer<TKey> comparer)`

Set Methods

Most of the LINQ set methods are supported in LINQ to Entities queries, with the exception of those that use an EqualityComparer<T>. For more information, see Standard Query Operators in LINQ to Entities Queries. The following table lists the supported and unsupported set methods.

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
All	Supported	`Function All(Of TSource) ( _ source As IQueryable(Of TSource), _ predicate As Expression(Of Func(Of TSource, Boolean)) _ ) As Boolean`	`bool All<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, bool>> predicate )`
Any	Supported	`Function Any(Of TSource) ( _ source As IQueryable(Of TSource) _ ) As Boolean`	`bool Any<TSource>( this IQueryable<TSource> source )`
Any	Supported	`Function Any(Of TSource) ( _ source As IQueryable(Of TSource), _ predicate As Expression(Of Func(Of TSource, Boolean)) _ ) As Boolean`	`bool Any<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, bool>> predicate )`
Contains	Supported	`Function Contains(Of TSource) ( _ source As IQueryable(Of TSource), _ item As TSource _ ) As Boolean`	`bool Contains<TSource> ( this IQueryable<TSource> source, TSource item )`
Contains	Not supported	`Function Contains(Of TSource) ( _ source As IQueryable(Of TSource), _ item As TSource, _ comparer As IEqualityComparer(Of TSource) _ ) As Boolean`	`bool Contains<TSource>( this IQueryable<TSource> source, TSource item, IEqualityComparer<TSource> comparer )`

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
Concat	Supported, but there is no guarantee of order being preserved	`Function Concat(Of TSource) ( _ source1 As IQueryable(Of TSource), _ source2 As IEnumerable(Of TSource) _ ) As IQueryable(Of TSource)`	`IQueryable<TSource> Concat<TSource>( this IQueryable<TSource> source1, IEnumerable<TSource> source2 )`
DefaultIfEmpty	Supported	`Function DefaultIfEmpty(Of TSource) ( _ source As IQueryable(Of TSource) _ ) As IQueryable(Of TSource)`	`IQueryable<TSource> DefaultIfEmpty<TSource> ( this IQueryable<TSource> source )`
DefaultIfEmpty	Supported	`Function DefaultIfEmpty(Of TSource) ( _ source As IQueryable(Of TSource), _ defaultValue As TSource _ ) As IQueryable(Of TSource)`	`IQueryable<TSource> DefaultIfEmpty<TSource> ( this IQueryable<TSource> source, TSource defaultValue )`
Distinct	Supported	`Function Distinct(Of TSource) ( _ source As IQueryable(Of TSource) _ ) As IQueryable(Of TSource)`	`IQueryable<TSource> Distinct<TSource>( this IQueryable<TSource> source )`
Distinct	Not supported	`Function Distinct(Of TSource) ( _ source As IQueryable(Of TSource), _ comparer As IEqualityComparer(Of TSource) _ ) As IQueryable(Of TSource)`	`IQueryable<TSource> Distinct<TSource>( this IQueryable<TSource> source, IEqualityComparer<TSource> comparer )`
Except	Supported	`Function Except(Of TSource) ( _ source1 As IQueryable(Of TSource), _ source2 As IEnumerable(Of TSource) _ ) As IQueryable(Of TSource)`	`IQueryable<TSource> Except<TSource>( this IQueryable<TSource> source1, IEnumerable<TSource> source2 )`
Except	Not supported	`Function Except(Of TSource) ( _ source1 As IQueryable(Of TSource), _ source2 As IEnumerable(Of TSource), _ comparer As IEqualityComparer(Of TSource) _ ) As IQueryable(Of TSource)`	`IQueryable<TSource> Except<TSource>( this IQueryable<TSource> source1, IEnumerable<TSource> source2, IEqualityComparer<TSource> comparer )`
Intersect	Supported	`Function Intersect(Of TSource) ( _ source1 As IQueryable(Of TSource), _ source2 As IEnumerable(Of TSource) _ ) As IQueryable(Of TSource)`	`IQueryable<TSource> Intersect<TSource>( this IQueryable<TSource> source1, IEnumerable<TSource> source2 )`

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
Intersect	Not supported	`Function Intersect(Of TSource) ( _ source1 As IQueryable(Of TSource), _ source2 As IEnumerable(Of TSource), _ comparer As IEqualityComparer(Of TSource) _ ) As IQueryable(Of TSource)`	`IQueryable<TSource> Intersect<TSource>( this IQueryable<TSource> source1, IEnumerable<TSource> source2, IEqualityComparer<TSource> comparer )`
Union	Supported	`Function Union(Of TSource) ( _ source1 As IQueryable(Of TSource), _ source2 As IEnumerable(Of TSource) _ ) As IQueryable(Of TSource)`	`IQueryable<TSource> Union<TSource>( this IQueryable<TSource> source1, IEnumerable<TSource> source2 )`
Union	Not supported	`Function Union(Of TSource) ( _ source1 As IQueryable(Of TSource), _ source2 As IEnumerable(Of TSource), _ comparer As IEqualityComparer(Of TSource) _ ) As IQueryable(Of TSource)`	`IQueryable<TSource> Union<TSource>( this IQueryable<TSource> source1, IEnumerable<TSource> source2, IEqualityComparer<TSource> comparer )`

Ordering Methods

Most of the LINQ ordering methods are supported in LINQ to Entities, with the exception of those that accept an IComparer<T>, because the comparer cannot be translated to the data source. For more information, see Standard Query Operators in LINQ to Entities Queries. The following table lists the supported and unsupported ordering methods.

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
OrderBy	Supported	`Function OrderBy(Of TSource, TKey) ( _ source As IQueryable(Of TSource), _ keySelector As Expression(Of Func(Of TSource, TKey)) _ ) As IOrderedQueryable(Of TSource)`	`IOrderedQueryable<TSource> OrderBy<TSource, TKey>( this IQueryable<TSource> source, Expression<Func<TSource, TKey>> keySelector )`
OrderBy	Not supported	`Function OrderBy(Of TSource, TKey) ( _ source As IQueryable(Of TSource), _ keySelector As Expression(Of Func(Of TSource, TKey)), _ comparer As IComparer(Of TKey) _ ) As IOrderedQueryable(Of TSource)`	`IOrderedQueryable<TSource> OrderBy\<TSource, TKey>( this IQueryable<TSource> source, Expression<Func\ <TSource, TKey>> keySelector, IComparer<TKey> comparer )`

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
OrderByDescending	Supported	`Function OrderByDescending(Of TSource, TKey) ( _ source As IQueryable(Of TSource), _ keySelector As Expression(Of Func(Of TSource, TKey)) _ ) As IOrderedQueryable(Of TSource)`	`IOrderedQueryable<TSource> OrderByDescending<TSource, TKey>( this IQueryable<TSource> source, Expression<Func<TSource, TKey>> keySelector )`
OrderByDescending	Not supported	`Function OrderByDescending(Of TSource, TKey) ( _ source As IQueryable(Of TSource), _ keySelector As Expression(Of Func(Of TSource, TKey)), _ comparer As IComparer(Of TKey) _ ) As IOrderedQueryable(Of TSource)`	`IOrderedQueryable<TSource> OrderByDescending\ <TSource, TKey>( this IQueryable<TSource> source, Expression<Func\ <TSource, TKey>> keySelector, IComparer<TKey> comparer )`
ThenBy	Supported	`Function ThenBy(Of TSource, TKey) ( _ source As IOrderedQueryable(Of TSource), _ keySelector As Expression(Of Func(Of TSource, TKey)) _ ) As IOrderedQueryable(Of TSource)`	`IOrderedQueryable<TSource> ThenBy<TSource, TKey>( this IOrderedQueryable<TSource> source, Expression<Func<TSource, TKey>> keySelector )`
ThenBy	Not supported	`Function ThenBy(Of TSource, TKey) ( _ source As IOrderedQueryable(Of TSource), _ keySelector As Expression(Of Func(Of TSource, TKey)), _ comparer As IComparer(Of TKey) _ ) As IOrderedQueryable(Of TSource)`	`IOrderedQueryable<TSource> ThenBy\<TSource, TKey>( this IOrderedQueryable<TSource> source, Expression<Func\ <TSource, TKey>> keySelector, IComparer<TKey> comparer )`
ThenByDescending	Supported	`Function ThenByDescending(Of TSource, TKey) ( _ source As IOrderedQueryable(Of TSource), _ keySelector As Expression(Of Func(Of TSource, TKey)) _ ) As IOrderedQueryable(Of TSource)`	`IOrderedQueryable<TSource> ThenByDescending<TSource, TKey>( this IOrderedQueryable<TSource> source, Expression<Func<TSource, TKey>> keySelector )`

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
ThenByDescending	Not supported	`Function ThenByDescending(Of TSource, TKey) ( _ source As IOrderedQueryable(Of TSource), _ keySelector As Expression(Of Func(Of TSource, TKey)), _ comparer As IComparer(Of TKey) _ ) As IOrderedQueryable(Of TSource)`	`IOrderedQueryable<TSource> ThenByDescending\<TSource, TKey>( this IOrderedQueryable<TSource> source, Expression<Func\ <TSource, TKey>> keySelector, IComparer<TKey> comparer )`
Reverse	Not supported	`Function Reverse(Of TSource) ( _ source As IQueryable(Of TSource) _ ) As IQueryable(Of TSource)`	`IQueryable<TSource> Reverse<TSource>( this IQueryable<TSource> source )`

Grouping Methods

Most of the LINQ grouping methods are supported in LINQ to Entities, with the exception of those that accept an IEqualityComparer<T>, because the comparer cannot be translated to the data source. For more information, see Standard Query Operators in LINQ to Entities Queries. The following table lists the supported and unsupported grouping methods.

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
GroupBy	Supported	`Function GroupBy(Of TSource, TKey) ( _ source As IQueryable(Of TSource), _ keySelector As Expression(Of Func(Of TSource, TKey)) _ ) As IQueryable(Of IGrouping(Of TKey, TSource))`	`IQueryable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>( this IQueryable<TSource> source, Expression<Func<TSource, TKey>> keySelector )`
GroupBy	Not supported	`Function GroupBy(Of TSource, TKey) ( _ source As IQueryable(Of TSource), _ keySelector As Expression(Of Func(Of TSource, TKey)), _ comparer As IEqualityComparer(Of TKey) _ ) As IQueryable(Of IGrouping(Of TKey, TSource))`	`IQueryable<IGrouping\ <TKey, TSource>> GroupBy\<TSource, TKey> ( this IQueryable<TSource> source, Expression<Func\ <TSource, TKey>> keySelector, IEqualityComparer<TKey> comparer )`

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
GroupBy	Supported	`Function GroupBy(Of TSource, TKey, TElement) ( _ source As IQueryable(Of TSource), _ keySelector As Expression(Of Func(Of TSource, TKey)), _ elementSelector As Expression(Of Func(Of TSource, TElement)) _ ) As IQueryable(Of IGrouping(Of TKey, TElement))`	`IQueryable<IGrouping<TKey, TElement>> GroupBy<TSource, TKey, TElement>( this IQueryable<TSource> source, Expression<Func<TSource, TKey>> keySelector, Expression<Func<TSource, TElement>> elementSelector )`
GroupBy	Supported	`Function GroupBy(Of TSource, TKey, TResult) ( _ source As IQueryable(Of TSource), _ keySelector As Expression(Of Func(Of TSource, TKey)), _ resultSelector As Expression(Of Func(Of TKey, IEnumerable(Of TSource), TResult)) _ ) As IQueryable(Of TResult)`	`IQueryable<TResult> GroupBy\<TSource, TKey, TResult>( this IQueryable<TSource> source, Expression<Func\ <TSource, TKey>> keySelector, Expression<Func<TKey, IEnumerable<TSource>, TResult>> resultSelector )`
GroupBy	Not supported	`Function GroupBy(Of TSource, TKey, TElement) ( _ source As IQueryable(Of TSource), _ keySelector As Expression(Of Func(Of TSource, TKey)), _ elementSelector As Expression(Of Func(Of TSource, TElement)), _ comparer As IEqualityComparer(Of TKey) _ ) As IQueryable(Of IGrouping(Of TKey, TElement))`	`IQueryable<IGrouping\ <TKey, TElement>> GroupBy\<TSource, TKey, TElement>( this IQueryable<TSource> source, Expression<Func\ <TSource, TKey>> keySelector, Expression<Func\ <TSource, TElement>> elementSelector, IEqualityComparer<TKey> comparer`
GroupBy	Supported	`Function GroupBy(Of TSource, TKey, TElement, TResult) (_ source As IQueryable(Of TSource), _ keySelector As Expression(Of Func(Of TSource, TKey)), _ elementSelector As Expression(Of Func(Of TSource, TElement)), _ resultSelector As Expression(Of Func(Of TKey, IEnumerable(Of TElement), TResult)) _) As IQueryable(Of TResult)`	`IQueryable<TResult> GroupBy<TSource, TKey, TElement, TResult>( this IQueryable<TSource> source, Expression<Func<TSource, TKey>> keySelector, Expression<Func<TSource, TElement>> elementSelector, Expression<Func<TKey, IEnumerable<TElement>, TResult>> resultSelector )`

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
GroupBy	Not supported	`Function GroupBy(Of TSource, TKey, TResult) ( _ source As IQueryable(Of TSource), _ keySelector As Expression(Of Func(Of TSource, TKey)), _ resultSelector As Expression(Of Func(Of TKey, IEnumerable(Of TSource), TResult)), _ comparer As IEqualityComparer(Of TKey) _ ) As IQueryable(Of TResult)`	`IQueryable<TResult> GroupBy\<TSource, TKey, TResult>( this IQueryable<TSource> source, Expression<Func\ <TSource, TKey>> keySelector, Expression<Func<TKey, IEnumerable<TSource>, TResult>> resultSelector, IEqualityComparer<TKey> comparer )`
GroupBy	Not supported	`Function GroupBy(Of TSource, TKey, TElement, TResult) (_ source As IQueryable(Of TSource), _ keySelector As Expression(Of Func(Of TSource, TKey)), _ elementSelector As Expression(Of Func(Of TSource, TElement)), _ resultSelector As Expression(Of Func(Of TKey, IEnumerable(Of TElement), TResult)), _ comparer As IEqualityComparer(Of TKey) _) As IQueryable(Of TResult)`	`IQueryable<TResult> GroupBy<TSource, TKey, TElement, TResult>( this IQueryable<TSource> source, Expression<Func<TSource, TKey>> keySelector, Expression<Func<TSource, TElement>> elementSelector, Expression<Func<TKey, IEnumerable<TElement>, TResult>> resultSelector, IEqualityComparer<TKey> comparer )`

Aggregate Methods

Most of the aggregate methods that accept primitive data types are supported in LINQ to Entities. For more information, see Standard Query Operators in LINQ to Entities Queries. The following table lists the supported and unsupported aggregate methods.

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
Aggregate	Not supported	`Function Aggregate(Of TSource) ( _ source As IQueryable(Of TSource), _ func As Expression(Of Func(Of TSource, TSource, TSource)) _ ) As TSource`	`TSource Aggregate<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, TSource, TSource>> func )`
Aggregate	Not supported	`Function Aggregate(Of TSource, TAccumulate) ( _ source As IQueryable(Of TSource), _ seed As TAccumulate, _ func As Expression(Of Func(Of TAccumulate, TSource, TAccumulate)) _ ) As TAccumulate`	`TAccumulate Aggregate<TSource, TAccumulate>( this IQueryable<TSource> source, TAccumulate seed, Expression<Func<TAccumulate, TSource, TAccumulate>> func )`

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
Aggregate	Not supported	`Function Aggregate(Of TSource, TAccumulate, TResult) ( _ source As IQueryable(Of TSource), _ seed As TAccumulate, _ func As Expression(Of Func(Of TAccumulate, TSource, TAccumulate)), _ selector As Expression(Of Func(Of TAccumulate, TResult)) _ ) As TResult`	`TResult Aggregate<TSource, TAccumulate, TResult>( this IQueryable<TSource> source, TAccumulate seed, Expression<Func<TAccumulate, TSource, TAccumulate>> func, Expression<Func<TAccumulate, TResult>> selector )`
Average	Supported	`Function Average ( _ source As IQueryable(Of Decimal) _ ) As Decimal`	`decimal Average( this IQueryable<decimal> source )`
Average	Supported	`Function Average ( _ source As IQueryable(Of Double) _ ) As Double`	`double Average( this IQueryable<double> source )`
Average	Supported	`Function Average ( _ source As IQueryable(Of Integer) _ ) As Double`	`double Average( this IQueryable<int> source )`
Average	Supported	`Function Average ( _ source As IQueryable(Of Long) _ ) As Double`	`double Average( this IQueryable<long> source )`
Average	Supported	`Function Average ( _ source As IQueryable(Of Nullable(Of Decimal)) _ ) As Nullable(Of Decimal)`	`Nullable<decimal> Average( this IQueryable<Nullable<decimal>> source )`
Average	Supported	`Function Average ( _ source As IQueryable(Of Nullable(Of Double)) _ ) As Nullable(Of Double)`	`Nullable<double> Average( this IQueryable<Nullable<double>> source )`
Average	Supported	`Function Average ( _ source As IQueryable(Of Nullable(Of Integer)) _ ) As Nullable(Of Double)`	`Nullable<double> Average( this IQueryable<Nullable<int>> source )`
Average	Supported	`Function Average ( _ source As IQueryable(Of Nullable(Of Long)) _ ) As Nullable(Of Double)`	`Nullable<double> Average( this IQueryable<Nullable<long>> source )`
Average	Supported	`Function Average ( _ source As IQueryable(Of Nullable(Of Single)) _ ) As Nullable(Of Single)`	`Nullable<float> Average( this IQueryable<Nullable<float>> source )`

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
Average	Supported	`Function Average ( _ source As IQueryable(Of Single) _ ) As Single`	`float Average( this IQueryable<float> source )`
Average	Not supported	`Function Average(Of TSource) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Integer)) _ ) As Double`	`double Average<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, int>> selector )`
Average	Not supported	`Function Average(Of TSource) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Nullable(Of Integer))) _ ) As Nullable(Of Double)`	`Nullable<double> Average<TSource>( this IQueryable<TSource> source, Expression<Func<TSource, Nullable<int>>> selector )`
Average	Not supported	`Function Average(Of TSource) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Long)) _ ) As Double`	`double Average<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, long>> selector )`
Average	Not supported	`Function Average(Of TSource) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Nullable(Of Long))) _ ) As Nullable(Of Double)`	`Nullable<double> Average<TSource>( this IQueryable<TSource> source, Expression<Func<TSource, Nullable<long>>> selector )`
Average	Not supported	`Function Average(Of TSource) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Single)) _ ) As Single`	`float Average<TSource> ( this IQueryable<TSource> source, Expression<Func\ <TSource, float>> selector )`
Average	Not supported	`Function Average(Of TSource) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Nullable(Of Single))) _ ) As Nullable(Of Single)`	`Nullable<float> Average<TSource>( this IQueryable<TSource> source, Expression<Func<TSource, Nullable<float>>> selector )`
Average	Not supported	`Function Average(Of TSource) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Double)) _ ) As Double`	`double Average<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, double>> selector )`

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
Average	Not supported	`Function Average(Of TSource) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Nullable(Of Double))) _ ) As Nullable(Of Double)`	`Nullable<double> Average<TSource>( this IQueryable<TSource> source, Expression<Func<TSource, Nullable<double>>> selector )`
Average	Not supported	`Function Average(Of TSource) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Decimal)) _ ) As Decimal`	`decimal Average<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, decimal>> selector )`
Average	Not supported	`Function Average(Of TSource) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Nullable(Of Decimal))) _ ) As Nullable(Of Decimal)`	`Nullable<decimal> Average<TSource>( this IQueryable<TSource> source, Expression<Func<TSource, Nullable<decimal>>> selector )`
Count	Supported	`Function Count(Of TSource) ( _ source As IQueryable(Of TSource) _ ) As Integer`	`int Count<TSource>( this IQueryable<TSource> source )`
Count	Not supported	`Function Count(Of TSource) ( _ source As IQueryable(Of TSource), _ predicate As Expression(Of Func(Of TSource, Boolean)) _ ) As Integer`	`int Count<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, bool>> predicate )`
LongCount	Supported	`Function LongCount(Of TSource) ( _ source As IQueryable(Of TSource) _ ) As Long`	`long LongCount<TSource>( this IQueryable<TSource> source )`
LongCount	Not supported	`Function LongCount(Of TSource) ( _ source As IQueryable(Of TSource), _ predicate As Expression(Of Func(Of TSource, Boolean)) _ ) As Long`	`long LongCount<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, bool>> predicate )`
Max	Supported	`Function Max(Of TSource) ( _ source As IQueryable(Of TSource) _ ) As TSource`	`TSource Max<TSource>( this IQueryable<TSource> source )`

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
Max	Not supported	`Function Max(Of TSource, TResult) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, TResult)) _ ) As TResult`	`TResult Max<TSource, TResult>( this IQueryable<TSource> source, Expression<Func<TSource, TResult>> selector )`
Min	Supported	`Function Min(Of TSource) ( _ source As IQueryable(Of TSource) _ ) As TSource`	`TSource Min<TSource>( this IQueryable<TSource> source )`
Min	Not supported	`Function Min(Of TSource, TResult) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, TResult)) _ ) As TResult`	`TResult Min<TSource, TResult>( this IQueryable<TSource> source, Expression<Func<TSource, TResult>> selector )`
Sum	Supported	`Function Sum ( _ source As IQueryable(Of Decimal) _ ) As Decimal`	`decimal Sum( this IQueryable<decimal> source )`
Sum	Supported	`Function Sum ( _ source As IQueryable(Of Double) _ ) As Double`	`double Sum( this IQueryable<double> source`
Sum	Supported	`Function Sum ( _ source As IQueryable(Of Integer) _ ) As Integer`	`int Sum( this IQueryable<int> source )`
Sum	Supported	`Function Sum ( _ source As IQueryable(Of Long) _ ) As Long`	`long Sum( this IQueryable<long> source )`
Sum	Supported	`Function Sum ( _ source As IQueryable(Of Nullable(Of Decimal)) _ ) As Nullable(Of Decimal)`	`Nullable<decimal> Sum( this IQueryable<Nullable<decimal>> source )`
Sum	Supported	`Function Sum ( _ source As IQueryable(Of Nullable(Of Double)) _ ) As Nullable(Of Double)`	`Function Sum ( _ source As IQueryable(Of Nullable(Of Double)) _ ) As Nullable(Of Double)Nullable<double> Sum( this IQueryable<Nullable<double>> source )`
Sum	Supported	`Function Sum ( _ source As IQueryable(Of Nullable(Of Integer)) _ ) As Nullable(Of Integer)`	`Nullable<int> Sum( this IQueryable<Nullable<int>> source )`

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
Sum	Supported	`Function Sum ( _ source As IQueryable(Of Nullable(Of Long)) _ ) As Nullable(Of Long)`	`Nullable<long> Sum( this IQueryable<Nullable<long>> source )`
Sum	Supported	`Function Sum ( _ source As IQueryable(Of Nullable(Of Single)) _ ) As Nullable(Of Single)`	`Nullable<float> Sum( this IQueryable<Nullable<float>> source )`
Sum	Supported	`Function Sum ( _ source As IQueryable(Of Single) _ ) As Single`	`float Sum( this IQueryable<float> source )`
Sum	Not supported	`Function Sum(Of TSource) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Integer)) _ ) As Integer`	`int Sum<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, int>> selector )`
Sum	Not supported	`Function Sum(Of TSource) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Nullable(Of Integer))) _ ) As Nullable(Of Integer)`	`Nullable<int> Sum<TSource>( this IQueryable<TSource> source, Expression<Func<TSource, Nullable<int>>> selector )`
Sum	Not supported	`Function Sum(Of TSource) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Long)) _ ) As Long`	`long Sum<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, long>> selector )`
Sum	Not supported	`Function Sum(Of TSource) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Nullable(Of Long))) _ ) As Nullable(Of Long)`	`Nullable<long> Sum<TSource>( this IQueryable<TSource> source, Expression<Func<TSource, Nullable<long>>> selector )`
Sum	Not supported	`Function Sum(Of TSource) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Nullable(Of Single))) _ ) As Nullable(Of Single)`	`Nullable<float> Sum<TSource>( this IQueryable<TSource> source, Expression<Func<TSource, Nullable<float>>> selector )`

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
Sum	Not supported	`Function Sum(Of TSource) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Single)) _ ) As Single`	`float Sum<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, float>> selector )`
Sum	Not supported	`Function Sum(Of TSource) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Double)) _ ) As Double`	`double Sum<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, double>> selector )`
Sum	Not supported	`Function Sum(Of TSource) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Nullable(Of Double))) _ ) As Nullable(Of Double)`	`Nullable<double> Sum<TSource>( this IQueryable<TSource> source, Expression<Func<TSource, Nullable<double>>> selector )`
Sum	Not supported	`Function Sum(Of TSource) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Decimal)) _ ) As Decimal`	`decimal Sum<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, decimal>> selector )`
Sum	Not supported	`Function Sum(Of TSource) ( _ source As IQueryable(Of TSource), _ selector As Expression(Of Func(Of TSource, Nullable(Of Decimal))) _ ) As Nullable(Of Decimal)`	`Nullable<decimal> Sum<TSource>( this IQueryable<TSource> source, Expression<Func<TSource, Nullable<decimal>>> selector )`

Type Methods

The LINQ standard query operators that deal with CLR type conversion and testing are supported in the Entity Framework. Only CLR types that map to conceptual model types are supported in LINQ to Entities. For a list of conceptual model types, see Conceptual Model Types (CSDL). The following table lists the supported and unsupported type methods.

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
Cast	Supported for EDM primitive types	`Function Cast(Of TResult) ( _ source As IQueryable _ ) As IQueryable(Of TResult)`	`IQueryable<TResult> Cast<TResult>( this IQueryable source )`
OfType	Supported for EntityType	`Function OfType(Of TResult) ( _ source As IQueryable _ ) As IQueryable(Of TResult)`	`IQueryable<TResult> OfType<TResult>( this IQueryable source )`

Paging Methods

A number of the LINQ paging methods are not supported in LINQ to Entities queries. For more information, see Standard Query Operators in LINQ to Entities Queries. The following table lists the supported and unsupported paging methods.

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
ElementAt	Not supported	`Function ElementAt(Of TSource) ( _ source As IQueryable(Of TSource), _ index As Integer _ ) As TSource`	`TSource ElementAt<TSource>( this IQueryable<TSource> source, int index )`
ElementAtOrDefault	Not supported	`Function ElementAtOrDefault(Of TSource) ( _ source As IQueryable(Of TSource), _ index As Integer _ ) As TSource`	`TSource ElementAtOrDefault<TSource> ( this IQueryable<TSource> source, int index )`
First	Supported	`Function First(Of TSource) ( _ source As IQueryable(Of TSource) _ ) As TSource`	`TSource First<TSource> ( this IQueryable<TSource> source )`
First	Supported	`Function First(Of TSource) ( _ source As IQueryable(Of TSource), _ predicate As Expression(Of Func(Of TSource, Boolean)) _ ) As TSource`	`TSource First<TSource> ( this IQueryable<TSource> source, Expression<Func\ <TSource, bool>> predicate )`
FirstOrDefault	Supported	`Function FirstOrDefault(Of TSource) ( _ source As IQueryable(Of TSource) _ ) As TSource`	`TSource FirstOrDefault<TSource> ( this IQueryable<TSource> source )`
FirstOrDefault	Supported	`Function FirstOrDefault(Of TSource) ( _ source As IQueryable(Of TSource), _ predicate As Expression(Of Func(Of TSource, Boolean)) _ ) As TSource`	`TSource FirstOrDefault<TSource> ( this IQueryable<TSource> source, Expression<Func\ <TSource, bool>> predicate )`
Last	Not supported	`Function Last(Of TSource) ( _ source As IQueryable(Of TSource) _ ) As TSource`	`TSource Last<TSource>( this IQueryable<TSource> source )`
Last	Not supported	`Function Last(Of TSource) ( _ source As IQueryable(Of TSource), _ predicate As Expression(Of Func(Of TSource, Boolean)) _ ) As TSource`	`TSource Last<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, bool>> predicate )`

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
LastOrDefault	Not supported	`Function LastOrDefault(Of TSource) ( _ source As IQueryable(Of TSource) _ ) As TSource`	`TSource LastOrDefault<TSource>( this IQueryable<TSource> source )`
LastOrDefault	Not supported	`Function LastOrDefault(Of TSource) ( _ source As IQueryable(Of TSource), _ predicate As Expression(Of Func(Of TSource, Boolean)) _ ) As TSource`	`TSource LastOrDefault<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, bool>> predicate )`
Single	Supported	`Function Single(Of TSource) ( _ source As IQueryable(Of TSource) _ ) As TSource`	`TSource Single<TSource>( this IQueryable<TSource> source )`
Single	Supported	`Function Single(Of TSource) ( _ source As IQueryable(Of TSource), _ predicate As Expression(Of Func(Of TSource, Boolean)) _ ) As TSource`	`TSource Single<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, bool>> predicate )`
SingleOrDefault	Supported	`Function SingleOrDefault(Of TSource) ( _ source As IQueryable(Of TSource) _ ) As TSource`	`TSource SingleOrDefault<TSource>( this IQueryable<TSource> source )`
SingleOrDefault	Supported	`Function SingleOrDefault(Of TSource) ( _ source As IQueryable(Of TSource), _ predicate As Expression(Of Func(Of TSource, Boolean)) _ ) As TSource`	`TSource SingleOrDefault<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, bool>> predicate )`
Skip	Supported	`Function Skip(Of TSource) ( _ source As IQueryable(Of TSource), _ count As Integer _ ) As IQueryable(Of TSource)`	`IQueryable<TSource> Skip<TSource>( this IQueryable<TSource> source, int count )`
SkipWhile	Not supported	`Function SkipWhile(Of TSource) ( _ source As IQueryable(Of TSource), _ predicate As Expression(Of Func(Of TSource, Boolean)) _ ) As IQueryable(Of TSource)`	`IQueryable<TSource> SkipWhile<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, bool>> predicate )`

METHOD	SUPPORT	VISUAL BASIC FUNCTION SIGNATURE	C# METHOD SIGNATURE
SkipWhile	Not supported	`Function SkipWhile(Of TSource) ( _ source As IQueryable(Of TSource), _ predicate As Expression(Of Func(Of TSource, Integer, Boolean)) _ ) As IQueryable(Of TSource)`	`IQueryable<TSource> SkipWhile<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, int, bool>> predicate )`
Take	Supported	`Function Take(Of TSource) ( _ source As IQueryable(Of TSource), _ count As Integer _ ) As IQueryable(Of TSource)`	`IQueryable<TSource> Take<TSource>( this IQueryable<TSource> source, int count )`
TakeWhile	Not supported	`Function TakeWhile(Of TSource) ( _ source As IQueryable(Of TSource), _ predicate As Expression(Of Func(Of TSource, Boolean)) _ ) As IQueryable(Of TSource)`	`IQueryable<TSource> TakeWhile<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, bool>> predicate )`
TakeWhile	Not supported	`Function TakeWhile(Of TSource) ( _ source As IQueryable(Of TSource), _ predicate As Expression(Of Func(Of TSource, Integer, Boolean)) _ ) As IQueryable(Of TSource)`	`IQueryable<TSource> TakeWhile<TSource>( this IQueryable<TSource> source, Expression<Func\ <TSource, int, bool>> predicate )`

See also

- Standard Query Operators in LINQ to Entities Queries

Known Issues and Considerations in LINQ to Entities

11/8/2022 • 4 minutes to read • Edit Online

This section provides information about known issues with LINQ to Entities queries.

LINQ Queries That cannot be Cached

Starting with .NET Framework 4.5, LINQ to Entities queries are automatically cached. However, LINQ to Entities queries that apply the `Enumerable.Contains` operator to in-memory collections are not automatically cached. Also parameterizing in-memory collections in compiled LINQ queries is not allowed.

Ordering Information Lost

Projecting columns into an anonymous type will cause ordering information to be lost in some queries that are executed against a SQL Server 2005 database set to a compatibility level of "80". This occurs when a column name in the order-by list matches a column name in the selector, as shown in the following example:

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    // Ordering information is lost when executed against a SQL Server 2005
    // database running with a compatibility level of "80".
    var results = context.Contacts.SelectMany(c => c.SalesOrderHeaders)
        .OrderBy(c => c.SalesOrderDetails.Count)
        .Select(c => new { c.SalesOrderDetails.Count });

    foreach (var result in results)
        Console.WriteLine(result.Count);
}
```

```
Using context As New AdventureWorksEntities()
    ' Ordering information is lost when executed against a SQL Server 2005
    ' database running with a compatibility level of "80".
    Dim results = context.Contacts.SelectMany(Function(c) c.SalesOrderHeaders) _
        .OrderBy(Function(c) c.SalesOrderDetails.Count) _
        .Select(Function(c) New With {c.SalesOrderDetails.Count})

    For Each result In results
        Console.WriteLine(result.Count)
    Next
End Using
```

Unsigned Integers Not Supported

Specifying an unsigned integer type in a LINQ to Entities query is not supported because the Entity Framework does not support unsigned integers. If you specify an unsigned integer, an ArgumentException exception will be thrown during the query expression translation, as shown in the following example. This example queries for an order with ID 48000.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    uint s = UInt32.Parse("48000");

    IQueryable<SalesOrderDetail> query = from sale in context.SalesOrderDetails
                                         where sale.SalesOrderID == s
                                         select sale;

    // NotSupportedException exception is thrown here.
    try
    {
        foreach (SalesOrderDetail order in query)
            Console.WriteLine("SalesOrderID: " + order.SalesOrderID);
    }
    catch (NotSupportedException ex)
    {
        Console.WriteLine("Exception: {0}", ex.Message);
    }
}
```

```
Using context As New AdventureWorksEntities()
    Dim saleId As UInteger = UInt32.Parse("48000")

    Dim query = _
        From sale In context.SalesOrderDetails _
        Where sale.SalesOrderID = saleId _
        Select sale

    Try
        ' NotSupportedException exception is thrown here.
        For Each order As SalesOrderDetail In query
            Console.WriteLine("SalesOrderID: " & order.SalesOrderID)
        Next
    Catch ex As NotSupportedException
        Console.WriteLine("Exception: " + ex.Message)
    End Try
End Using
```

Type Conversion Errors

In Visual Basic, when a property is mapped to a column of SQL Server bit type with a value of 1 using the `CByte` function, a SqlException is thrown with an "Arithmetic overflow error" message. The following example queries the `Product.MakeFlag` column in the AdventureWorks sample database and an exception is thrown when the query results are iterated over.

```
Using context As New AdventureWorksEntities()
    Dim productsList = _
        From product In context.Products _
        Select CByte(product.MakeFlag)

    ' Throws an SqlException exception with a "Arithmetic overflow error
    ' for data type tinyint" message when a value of 1 is iterated over.
    For Each makeFlag In productsList
        Console.WriteLine(makeFlag)
    Next
End Using
```

Referencing Non-Scalar Variables Not Supported

Referencing a non-scalar variables, such as an entity, in a query is not supported. When such a query executes, a NotSupportedException exception is thrown with a message that states "Unable to create a constant value of type `EntityType`. Only primitive types ('such as Int32, String, and Guid') are supported in this context."

> **NOTE**
> Referencing a collection of scalar variables is supported.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    Contact contact = context.Contacts.FirstOrDefault();

    // Referencing a non-scalar closure in a query will
    // throw an exception when the query is executed.
    IQueryable<string> contacts = from c in context.Contacts
        where c == contact
        select c.LastName;

    try
    {
        foreach (string name in contacts)
        {
            Console.WriteLine("Name: ", name);
        }
    }
    catch (NotSupportedException ex)
    {
        Console.WriteLine(ex.Message);
    }
}
```

```
Using context As New AdventureWorksEntities()

    Dim contact As Contact = context.Contacts.FirstOrDefault()

    ' Referencing a non-scalar closure in a query will
    ' throw an exception when the query is executed.
    Dim contacts = From c In context.Contacts _
                   Where c.Equals(contact) _
                   Select c.LastName

    Try
        For Each name As String In contacts
            Console.WriteLine("Name: ", name)
        Next

    Catch ex As Exception
        Console.WriteLine(ex.Message)
    End Try

End Using
```

Nested Queries May Fail with SQL Server 2000

With SQL Server 2000, LINQ to Entities queries may fail if they produce nested Transact-SQL queries that are three or more levels deep.

Projecting to an Anonymous Type

If you define your initial query path to include related objects by using the Include method on the ObjectQuery<T> and then use LINQ to project the returned objects to an anonymous type, the objects specified in the include method are not included in the query results.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var resultWithoutRelatedObjects =
        context.Contacts.Include("SalesOrderHeaders").Select(c => new { c }).FirstOrDefault();
    if (resultWithoutRelatedObjects.c.SalesOrderHeaders.Count == 0)
    {
        Console.WriteLine("No orders are included.");
    }
}
```

```
Using context As New AdventureWorksEntities()
    Dim resultWithoutRelatedObjects = context.Contacts. _
        Include("SalesOrderHeaders"). _
        Select(Function(c) New With {c}).FirstOrDefault()
    If resultWithoutRelatedObjects.c.SalesOrderHeaders.Count = 0 Then
        Console.WriteLine("No orders are included.")
    End If
End Using
```

To get related objects, do not project returned types to an anonymous type.

```
using (AdventureWorksEntities context = new AdventureWorksEntities())
{
    var resultWithRelatedObjects =
        context.Contacts.Include("SalesOrderHeaders").Select(c => c).FirstOrDefault();
    if (resultWithRelatedObjects.SalesOrderHeaders.Count != 0)
    {
        Console.WriteLine("Orders are included.");
    }
}
```

```
Using context As New AdventureWorksEntities()
    Dim resultWithRelatedObjects = context.Contacts. _
        Include("SalesOrderHeaders"). _
        Select(Function(c) c).FirstOrDefault()
    If resultWithRelatedObjects.SalesOrderHeaders.Count <> 0 Then
        Console.WriteLine("Orders are included.")
    End If
End Using
```

See also

- LINQ to Entities

Entity SQL Language

11/8/2022 • 2 minutes to read • Edit Online

Entity SQL is a storage-independent query language that is similar to SQL. Entity SQL allows you to query entity data, either as objects or in a tabular form. You should consider using Entity SQL in the following cases:

- When a query must be dynamically constructed at run time. In this case, you should also consider using the query builder methods of ObjectQuery<T> instead of constructing an Entity SQL query string at run time.
- When you want to define a query as part of the model definition. Only Entity SQL is supported in a data model. For more information, see QueryView Element (MSL)
- When using EntityClient to return read-only entity data as rowsets using a EntityDataReader. For more information, see EntityClient Provider for the Entity Framework.
- If you are already an expert in SQL-based query languages, Entity SQL may seem the most natural to you.

Using Entity SQL with the EntityClient provider

If you want to use Entity SQL with the EntityClient provider, see the following topics for more information:

EntityClient Provider for the Entity Framework

How to: Build an EntityConnection Connection String

How to: Execute a Query that Returns PrimitiveType Results

How to: Execute a Query that Returns StructuralType Results

How to: Execute a Query that Returns RefType Results

How to: Execute a Query that Returns Complex Types

How to: Execute a Query that Returns Nested Collections

How to: Execute a Parameterized Entity SQL Query Using EntityCommand

How to: Execute a Parameterized Stored Procedure Using EntityCommand

How to: Execute a Polymorphic Query

How to: Navigate Relationships with the Navigate Operator

Using Entity SQL with object queries

If you want to use Entity SQL with object queries, see the following topics for more information:

How to: Execute a Query that Returns Entity Type Objects

How to: Execute a Parameterized Query

How to: Navigate Relationships Using Navigation Properties

How to: Call a User-Defined Function

How to: Filter Data

How to: Sort Data

How to: Group Data

How to: Aggregate Data

How to: Execute a Query that Returns Anonymous Type Objects

How to: Execute a Query that Returns a Collection of Primitive Types

How to: Query Related Objects in an EntityCollection

How to: Order the Union of Two Queries

How to: Page Through Query Results

In This Section

Entity SQL Overview

Entity SQL Reference

See also

- ADO.NET Entity Framework
- Language Reference

Entity SQL Overview

11/8/2022 • 2 minutes to read • Edit Online

Entity SQL is a SQL-like language that enables you to query conceptual models in the Entity Framework. Conceptual models represent data as entities and relationships, and Entity SQL allows you to query those entities and relationships in a format that is familiar to those who have used SQL.

The Entity Framework works with storage-specific data providers to translate generic Entity SQL into storage-specific queries. The EntityClient provider supplies a way to execute an Entity SQL command against an entity model and return rich types of data including scalar results, result sets, and object graphs. When you construct EntityCommand objects, you can specify a stored procedure name or the text of a query by assigning an Entity SQL query string to its EntityCommand.CommandText property. The EntityDataReader exposes the results of executing a EntityCommand against an EDM. To execute the command that returns the EntityDataReader, call ExecuteReader.

In addition to the EntityClient provider, the Entity Framework enables you to use Entity SQL to execute queries against a conceptual model and return data as strongly typed CLR objects that are instances of entity types. For more information, see Working with Objects.

This section provides conceptual information about Entity SQL.

In This Section

Paging

Comparison Semantics

Composing Nested Entity SQL Queries

Nullable Structured Types

See also

- Entity SQL Reference
- Entity SQL Language
- CSDL, SSDL, and MSL Specifications

How Entity SQL differs from Transact-SQL

11/8/2022 • 7 minutes to read • Edit Online

This article describes the differences between Entity SQL and Transact-SQL.

Inheritance and Relationships Support

Entity SQL works directly with conceptual entity schemas and supports conceptual model features such as inheritance and relationships.

When working with inheritance, it is often useful to select instances of a subtype from a collection of supertype instances. The oftype operator in Entity SQL (similar to `oftype` in C# Sequences) provides this capability.

Support for Collections

Entity SQL treats collections as first-class entities. For example:

- Collection expressions are valid in a `from` clause.
- `in` and `exists` subqueries have been generalized to allow any collections.

  A subquery is one kind of collection. `e1 in e2` and `exists(e)` are the Entity SQL constructs to perform these operations.
- Set operations, such as `union`, `intersect`, and `except`, now operate on collections.
- Joins operate on collections.

Support for Expressions

Transact-SQL has subqueries (tables) and expressions (rows and columns).

To support collections and nested collections, Entity SQL makes everything an expression. Entity SQL is more composable than Transact-SQL—every expression can be used anywhere. Query expressions always result in collections of the projected types and can be used anywhere a collection expression is allowed. For information about Transact-SQL expressions that are not supported in Entity SQL, see Unsupported Expressions.

The following are all valid Entity SQL queries:

```
1+2 *3
"abc"
row(1 as a, 2 as b)
{ 1, 3, 5}
e1 union all e2
set(e1)
```

Uniform Treatment of Subqueries

Given its emphasis on tables, Transact-SQL performs contextual interpretation of subqueries. For example, a subquery in the `from` clause is considered to be a multiset (table). But the same subquery used in the `select` clause is considered to be a scalar subquery. Similarly, a subquery used on the left side of an `in` operator is considered to be a scalar subquery, while the right side is expected to be a multiset subquery.

Entity SQL eliminates these differences. An expression has a uniform interpretation that does not depend on the context in which it is used. Entity SQL considers all subqueries to be multiset subqueries. If a scalar value is desired from the subquery, Entity SQL provides the `anyelement` operator that operates on a collection (in this case, the subquery), and extracts a singleton value from the collection.

Avoiding Implicit Coercions for Subqueries

A related side effect of uniform treatment of subqueries is implicit conversion of subqueries to scalar values. Specifically, in Transact-SQL, a multiset of rows (with a single field) is implicitly converted into a scalar value whose data type is that of the field.

Entity SQL does not support this implicit coercion. Entity SQL provides the `ANYELEMENT` operator to extract a singleton value from a collection, and a `select value` clause to avoid creating a row-wrapper during a query expression.

Select Value: Avoiding the Implicit Row Wrapper

The select clause in a Transact-SQL subquery implicitly creates a row wrapper around the items in the clause. This implies that we cannot create collections of scalars or objects. Transact-SQL allows an implicit coercion between a `rowtype` with one field and a singleton value of the same data type.

Entity SQL provides the `select value` clause to skip the implicit row construction. Only one item may be specified in a `select value` clause. When such a clause is used, no row wrapper is constructed around the items in the `select` clause, and a collection of the desired shape may be produced, for example, `select value a`.

Entity SQL also provides the row constructor to construct arbitrary rows. `select` takes one or more elements in the projection and results in a data record with fields:

`select a, b, c`

Left Correlation and Aliasing

In Transact-SQL, expressions in a given scope (a single clause like `select` or `from`) cannot reference expressions defined earlier in the same scope. Some dialects of SQL (including Transact-SQL) do support limited forms of these in the `from` clause.

Entity SQL generalizes left correlations in the `from` clause, and treats them uniformly. Expressions in the `from` clause can reference earlier definitions (definitions to the left) in the same clause without the need for additional syntax.

Entity SQL also imposes additional restrictions on queries involving `group by` clauses. Expressions in the `select` clause and `having` clause of such queries may only refer to the `group by` keys via their aliases. The following construct is valid in Transact-SQL but are not in Entity SQL:

```
SELECT t.x + t.y FROM T AS t group BY t.x + t.y
```

To do this in Entity SQL:

```
SELECT k FROM T AS t GROUP BY (t.x + t.y) AS k
```

Referencing Columns (Properties) of Tables (Collections)

All column references in Entity SQL must be qualified with the table alias. The following construct (assuming that `a` is a valid column of table `T`) is valid in Transact-SQL but not in Entity SQL.

```
SELECT a FROM T
```

The Entity SQL form is

```
SELECT t.a AS A FROM T AS t
```

The table aliases are optional in the `from` clause. The name of the table is used as the implicit alias. Entity SQL allows the following form as well:

```
SELECT Tab.a FROM Tab
```

Navigation Through Objects

Transact-SQL uses the "." notation for referencing columns of (a row of) a table. Entity SQL extends this notation (borrowed from programming languages) to support navigation through properties of an object.

For example, if `p` is an expression of type Person, the following is the Entity SQL syntax for referencing the city of the address of this person.

```
p.Address.City
```

No Support for *

Transact-SQL supports the unqualified * syntax as an alias for the entire row, and the qualified * syntax (t.*) as a shortcut for the fields of that table. In addition, Transact-SQL allows for a special count(*) aggregate, which includes nulls.

Entity SQL does not support the * construct. Transact-SQL queries of the form `select * from T` and `select T1.* from T1, T2...` can be expressed in Entity SQL as `select value t from T as t` and `select value t1 from T1 as t1, T2 as t2...`, respectively. Additionally, these constructs handle inheritance (value substitutability), while the `select *` variants are restricted to top-level properties of the declared type.

Entity SQL does not support the `count(*)` aggregate. Use `count(0)` instead.

Changes to Group By

Entity SQL supports aliasing of `group by` keys. Expressions in the `select` clause and `having` clause must refer to the `group by` keys via these aliases. For example, this Entity SQL syntax:

```
SELECT k1, count(t.a), sum(t.a)
FROM T AS t
GROUP BY t.b + t.c AS k1
```

...is equivalent to the following Transact-SQL:

```
SELECT b + c, count(*), sum(a)
FROM T
GROUP BY b + c
```

Collection-Based Aggregates

Entity SQL supports two kinds of aggregates.

Collection-based aggregates operate on collections and produce the aggregated result. These can appear anywhere in the query, and do not require a `group by` clause. For example:

```
SELECT t.a AS a, count({1,2,3}) AS b FROM T AS t
```

Entity SQL also supports SQL-style aggregates. For example:

```
SELECT a, sum(t.b) FROM T AS t GROUP BY t.a AS a
```

ORDER BY Clause Usage

Transact-SQL allows `ORDER BY` clauses to be specified only in the topmost `SELECT .. FROM .. WHERE` block. In Entity SQL, you can use a nested `ORDER BY` expression and it can be placed anywhere in the query, but ordering in a nested query is not preserved.

```
-- The following query will order the results by the last name
SELECT C1.FirstName, C1.LastName
        FROM AdventureWorks.Contact AS C1
        ORDER BY C1.LastName
```

```
-- In the following query ordering of the nested query is ignored.
SELECT C2.FirstName, C2.LastName
    FROM (SELECT C1.FirstName, C1.LastName
        FROM AdventureWorks.Contact as C1
        ORDER BY C1.LastName) as C2
```

Identifiers

In Transact-SQL, identifier comparison is based on the collation of the current database. In Entity SQL, identifiers are always case insensitive and accent sensitive (that is, Entity SQL distinguishes between accented and unaccented characters; for example, 'a' is not equal to 'ã'). Entity SQL treats versions of letters that appear the same but are from different code pages as different characters. For more information, see Input Character Set.

Transact-SQL Functionality Not Available in Entity SQL

The following Transact-SQL functionality is not available in Entity SQL.

DML
Entity SQL currently provides no support for DML statements (insert, update, delete).

DDL
Entity SQL provides no support for DDL in the current version.

Imperative Programming
Entity SQL provides no support for imperative programming, unlike Transact-SQL. Use a programming language instead.

Grouping Functions
Entity SQL does not yet provide support for grouping functions (for example, CUBE, ROLLUP, and GROUPING_SET).

Analytic Functions

Entity SQL does not (yet) provide support for analytic functions.

Built-in Functions, Operators
Entity SQL supports a subset of Transact-SQL's built in functions and operators. These operators and functions are likely to be supported by the major store providers. Entity SQL uses the store-specific functions declared in a provider manifest. Additionally, the Entity Framework allows you to declare built-in and user-defined existing store functions, for Entity SQL to use.

Hints
Entity SQL does not provide mechanisms for query hints.

Batching Query Results
Entity SQL does not support batching query results. For example, the following is valid Transact-SQL (sending as a batch):

```
SELECT * FROM products;
SELECT * FROM catagories;
```

However, the equivalent Entity SQL is not supported:

```
SELECT value p FROM Products AS p;
SELECT value c FROM Categories AS c;
```

Entity SQL only supports one result-producing query statement per command.

See also

- Entity SQL Overview
- Unsupported Expressions

Entity SQL Quick Reference

11/8/2022 • 4 minutes to read • Edit Online

This topic provides a quick reference to Entity SQL queries. The queries in this topic are based on the AdventureWorks Sales model.

Literals

String

There are Unicode and non-Unicode character string literals. Unicode strings are prepended with N. For example, `N'hello'`.

The following is an example of a Non-Unicode string literal:

```
'hello'
--same as
"hello"
```

Output:

VALUE
hello

DateTime

In DateTime literals, both date and time parts are mandatory. There are no default values.

Example:

```
DATETIME '2006-12-25 01:01:00.000'
--same as
DATETIME '2006-12-25 01:01'
```

Output:

VALUE
12/25/2006 1:01:00 AM

Integer

Integer literals can be of type Int32 (123), UInt32 (123U), Int64 (123L), and UInt64 (123UL).

Example:

```
--a collection of integers
{1, 2, 3}
```

Output:

VALUE
1
2
3

Other

Other literals supported by Entity SQL are Guid, Binary, Float/Double, Decimal, and `null`. Null literals in Entity SQL are considered to be compatible with every other type in the conceptual model.

Type Constructors

ROW

ROW constructs an anonymous, structurally-typed (record) value as in: `ROW(1 AS myNumber, 'Name' AS myName).`

Example:

```
SELECT VALUE row (product.ProductID AS ProductID, product.Name
    AS ProductName) FROM AdventureWorksEntities.Product AS product
```

Output:

PRODUCTID	NAME
1	Adjustable Race
879	All-Purpose Bike Stand
712	AWC Logo Cap
...	...

MULTISET

MULTISET constructs collections, such as:

`MULTISET(1,2,2,3)` `--same as` - `{1,2,2,3}.`

Example:

```
SELECT VALUE product FROM AdventureWorksEntities.Product AS product WHERE product.ListPrice IN MultiSet
(125, 300)
```

Output:

PRODUCTID	NAME	PRODUCTNUMBER	...
842	Touring-Panniers, Large	PA-T100	...

Object

Named Type Constructor constructs (named) user-defined objects, such as `person("abc", 12)`.

Example:

```
SELECT VALUE AdventureWorksModel.SalesOrderDetail (o.SalesOrderDetailID, o.CarrierTrackingNumber,
o.OrderQty,
o.ProductID, o.SpecialOfferID, o.UnitPrice, o.UnitPriceDiscount,
o.rowguid, o.ModifiedDate) FROM AdventureWorksEntities.SalesOrderDetail
AS o
```

Output:

SALESORDERDETAILID	CARRIERTRACKINGNUMBER	ORDERQTY	PRODUCTID	...
1	4911-403C-98	1	776	...
2	4911-403C-98	3	777	...
...	...	...	...	...

References

REF

REF creates a reference to an entity type instance. For example, the following query returns references to each Order entity in the Orders entity set:

```
SELECT REF(o) AS OrderID FROM Orders AS o
```

Output:

VALUE
1
2
3
...

The following example uses the property extraction operator (.) to access a property of an entity. When the property extraction operator is used, the reference is automatically dereferenced.

Example:

```
SELECT VALUE REF(p).Name FROM
    AdventureWorksEntities.Product AS p
```

Output:

VALUE
Adjustable Race

VALUE
All-Purpose Bike Stand
AWC Logo Cap
...

DEREF

DEREF dereferences a reference value and produces the result of that dereference. For example, the following query produces the Order entities for each Order in the Orders entity set: `SELECT DEREF(o2.r) FROM (SELECT REF(o) AS r FROM LOB.Orders AS o) AS o2`..

Example:

```
SELECT VALUE DEREF(REF(p)).Name FROM
    AdventureWorksEntities.Product AS p
```

Output:

VALUE
Adjustable Race
All-Purpose Bike Stand
AWC Logo Cap
...

CREATEREF AND KEY

CREATEREF creates a reference passing a key. KEY extracts the key portion of an expression with type reference.

Example:

```
SELECT VALUE Key(CreateRef(AdventureWorksEntities.Product, row(p.ProductID)))
    FROM AdventureWorksEntities.Product AS p
```

Output:

PRODUCTID
980
365
771
...

Functions

Canonical

The namespace for canonical functions is Edm, as in `Edm.Length("string")`. You do not have to specify the namespace unless another namespace is imported that contains a function with the same name as a canonical function. If two namespaces have the same function, the user should specify the full name.

Example:

```
SELECT Length(c. FirstName) AS NameLen FROM
    AdventureWorksEntities.Contact AS c
    WHERE c.ContactID BETWEEN 10 AND 12
```

Output:

NAMELEN
6
6
5

Microsoft Provider-Specific

Microsoft provider-specific functions are in the `SqlServer` namespace.

Example:

```
SELECT SqlServer.LEN(c.EmailAddress) AS EmailLen FROM
    AdventureWorksEntities.Contact AS c WHERE
    c.ContactID BETWEEN 10 AND 12
```

Output:

EMAILLEN
27
27
26

Namespaces

USING specifies namespaces used in a query expression.

Example:

```
using SqlServer; LOWER('AA');
```

Output:

VALUE

VALUE
aa

Paging

Paging can be expressed by declaring a SKIP and LIMIT sub-clauses to the ORDER BY clause.

Example:

```
SELECT c.ContactID as ID, c.LastName AS Name FROM
    AdventureWorks.Contact AS c ORDER BY c.ContactID SKIP 9 LIMIT 3;
```

Output:

ID	NAME
10	Adina
11	Agcaoili
12	Aguilar

Grouping

GROUPING BY specifies groups into which objects returned by a query (SELECT) expression are to be placed.

Example:

```
SELECT VALUE name FROM AdventureWorksEntities.Product AS P
    GROUP BY P.Name HAVING MAX(P.ListPrice) > 5
```

Output:

NAME
LL Mountain Seat Assembly
ML Mountain Seat Assembly
HL Mountain Seat Assembly
...

Navigation

The relationship navigation operator allows you to navigate over the relationship from one entity (from end) to another (to end). NAVIGATE takes the relationship type qualified as <namespace>.<relationship type name>. Navigate returns Ref<T> if the cardinality of the to end is 1. If the cardinality of the to end is n, the Collection<Ref<T>> will be returned.

Example:

```
SELECT a.AddressID, (SELECT VALUE DEREF(v) FROM
    NAVIGATE(a, AdventureWorksModel.FK_SalesOrderHeader_Address_BillToAddressID) AS v)
    FROM AdventureWorksEntities.Address AS a
```

Output:

ADDRESSID
1
2
3
...

SELECT VALUE AND SELECT

SELECT VALUE

Entity SQL provides the SELECT VALUE clause to skip the implicit row construction. Only one item can be specified in a SELECT VALUE clause. When such a clause is used, no row wrapper is constructed around the items in the SELECT clause, and a collection of the desired shape can be produced, for example: `SELECT VALUE a`.

Example:

```
SELECT VALUE p.Name FROM AdventureWorksEntities.Product AS p
```

Output:

NAME
Adjustable Race
All-Purpose Bike Stand
AWC Logo Cap
...

SELECT

Entity SQL also provides the row constructor to construct arbitrary rows. SELECT takes one or more elements in the projection and results in a data record with fields, for example: `SELECT a, b, c`.

Example:

SELECT p.Name, p.ProductID FROM AdventureWorksEntities.Product as p Output:

NAME	PRODUCTID
Adjustable Race	1
All-Purpose Bike Stand	879

NAME	PRODUCTID
AWC Logo Cap	712
...	...

CASE EXPRESSION

The case expression evaluates a set of Boolean expressions to determine the result.

Example:

```
CASE WHEN AVG({25,12,11}) < 100 THEN TRUE ELSE FALSE END
```

Output:

VALUE
TRUE

See also

- Entity SQL Reference
- Entity SQL Overview

Type System (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Entity SQL supports a number of types:

- Primitive (simple) types such as `Int32` and `String.`
- Nominal types that are defined in the schema, such as EntityType, ComplexType, and RelationshipType.
- Anonymous types that are not defined in the schema explicitly: CollectionType, RowType, and RefType.

This section discusses the anonymous types that are not defined in the schema explicitly but are supported by Entity SQL. For information on primitive and nominal types, see Conceptual Model Types (CSDL).

Rows

The structure of a row depends on the sequence of typed and named members that the row consists of. A row type has no identity and cannot be inherited from. Instances of the same row type are equivalent if the members are respectively equivalent. Rows have no behavior beyond their structural equivalence and have no equivalent in the common language runtime. Queries can result in structures that contain rows or collections of rows. The API binding between the Entity SQL queries and the host language defines how rows are realized in the query that produced the result. For information on how to construct a row instance, see Constructing Types.

Collections

Collection types represent zero or more instances of other objects. For information on how to construct collection, see Constructing Types.

References

A reference is a logical pointer to a specific entity in a specific entity set.

Entity SQL supports the following operators to construct, deconstruct, and navigate through references:

- REF
- CREATEREF
- KEY
- DEREF

You can navigate through a reference by using the member access (dot) operator(`.`). The following snippet extracts the Id property (of Order) by navigating through the r (reference) property.

```
select o2.r.Id
from (select ref(o) as r from LOB.Orders as o) as o2
```

If the reference value is null, or if the target of the reference does not exist, the result is null.

See also

- Entity SQL Overview

- Entity SQL Reference
- CAST
- CSDL, SSDL, and MSL Specifications

Type Definitions (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

A type definition is used in the declaration statement of an Entity SQL Inline function.

Remarks

The declaration statement for an inline function consists of the FUNCTION keyword followed by the identifier representing the function name (for example, "MyAvg") followed by a parameter definition list in parenthesis (for example, "dues Collection(Decimal)").

The parameter definition list consists of zero or more parameter definitions. Each parameter definition consists of an identifier (the name of the parameter to the function, for example, "dues") followed by a type definition (for example, "Collection(Decimal)").

The type definitions can be either:

- The type of the identifier (for example, "Int32" or "AdventureWorks.Order").
- The keyword `COLLECTION` followed by another type definition in parenthesis (for example, "Collection(AdventureWorks.Order)").
- The keyword ROW followed by a list of property definitions in parenthesis (for example, "Row(x AdventureWorks.Order)"). Property definitions have a format such as "`identifier type_definition`, `identifier type_definition`, ...".
- The keyword REF followed by the type of the identifier in parenthesis (for example, "Ref(AdventureWorks.Order)"). The REF type definition operator requires an entity type as the argument. You cannot specify a primitive type as the argument.

You can also nest type definitions (for example, "Collection(Row(x Ref(AdventureWorks.Order)))").

The type definition options are:

- `IdentifierName supported_type`, or
- `IdentifierName` COLLECTION(`type_definition`), or
- `IdentifierName` ROW(`property_definition`), or
- `IdentifierName` REF(`supported_entity_type`)

The property definition option is `IdentifierName type_definition`.

Supported types are any types in the current namespace. These include both primitive and entity types.

Supported entity types refer to only entity types in the current namespace. They do not include primitive types.

Examples

The following is an example of a simple type definition.

```
USING Microsoft.Samples.Entity
Function MyRound(p1 EDM.Decimal) AS (
   Round(p1)
)
MyRound(CAST(1.7 as EDM.Decimal))
```

The following is an example of a COLLECTION type definition.

```
USING Microsoft.Samples.Entity
Function MyRound(p1 Collection(EDM.Decimal)) AS (
   Select Round(p1) from p1
)
MyRound({CAST(1.7 as EDM.Decimal), CAST(2.7 as EDM.Decimal)})
```

The following is an example of a ROW type definition.

```
USING Microsoft.Samples.Entity
Function MyRound(p1 Row(x EDM.Decimal)) AS (
   Round(p1.x)
)
select MyRound(row(a as x)) from {CAST(1.7 as EDM.Decimal), CAST(2.7 as EDM.Decimal)} as a
```

The following is an example of a REF type definition.

```
USING Microsoft.Samples.Entity
Function UnReference(p1 Ref(AdventureWorks.Order)) AS (
   Deref(p1)
)
select Ref(x) from AdventureWorksEntities.SalesOrderHeaders as x
```

See also

- Entity SQL Overview
- Entity SQL Reference

Constructing Types (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Entity SQL provides three kinds of constructors: row constructors, named type constructors, and collection constructors.

Row Constructors

You use row constructors in Entity SQL to construct anonymous, structurally typed records from one or more values. The result type of a row constructor is a row type whose field types correspond to the types of the values used to construct the row. For example, the following expression constructs a value of type `Record(a int, b string, c int)`:

```
ROW(1 AS a, "abc" AS b, a + 34 AS c)
```

If you do not provide an alias for an expression in a row constructor, the Entity Framework will try to generate one. For more information, see the "Aliasing Rules" section in Identifiers.

The following rules apply to expression aliasing in a row constructor:

- Expressions in a row constructor cannot refer to other aliases in the same constructor.
- Two expressions in the same row constructor cannot have the same alias.

For more information about row constructors, see ROW.

Collection Constructors

You use collection constructors in Entity SQL to create an instance of a multiset from a list of values. All the values in the constructor must be of mutually compatible type `T`, and the constructor produces a collection of type `Multiset<T>`. For example, the following expression creates a collection of integers:

```
Multiset(1, 2, 3)
```

```
{1, 2, 3}
```

Empty multiset constructors are not allowed because the type of the elements cannot be determined. The following is not valid:

```
multiset() {}
```

For more information, see MULTISET.

Named Type Constructors (NamedType Initializers)

Entity SQL allows type constructors (initializers) to create instances of named complex types and entity types. For example, the following expression creates an instance of a `Person` type.

```
Person("abc", 12)
```

The following expression creates an instance of a complex type.

```
MyModel.ZipCode('98118', '4567')
```

The following expression creates an instance of a nested complex type.

```
MyModel.AddressInfo('My street address', 'Seattle', 'WA', MyModel.ZipCode('98118', '4567'))
```

The following expression creates an instance of an entity with a nested complex type.

```
MyModel.Person("Bill", MyModel.AddressInfo('My street address', 'Seattle', 'WA', MyModel.ZipCode('98118',
'4567')))
```

The following example shows how to initialize a property of a complex type to null.

```
MyModel.ZipCode('98118', null)
```

The arguments to the constructor are assumed to be in the same order as the declaration of the attributes of the type.

For more information, see Named Type Constructor.

See also

- Entity SQL Reference
- Entity SQL Overview
- Type System

Query Plan Caching (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Whenever an attempt to execute a query is made, the query pipeline looks up its query plan cache to see whether the exact query is already compiled and available. If so, it reuses the cached plan rather than building a new one. If a match is not found in the query plan cache, the query is compiled and cached. A query is identified by its Entity SQL text and parameter collection (names and types). All text comparisons are case-sensitive.

Configuration

Query plan caching is configurable through the EntityCommand.

To enable or disable query plan caching through EntityCommand.EnablePlanCaching, set this property to `true` or `false`. Disabling plan caching for individual dynamic queries that are unlikely to be used more then once improves performance.

You can enable query plan caching through EnablePlanCaching.

Recommended Practice

Dynamic queries should be avoided, in general. The following dynamic query example is vulnerable to SQL injection attacks, because it takes user input directly without any validation.

```
var query = "SELECT sp.SalesYTD FROM AdventureWorksEntities.SalesPerson as sp WHERE sp.EmployeeID = " +
employeeTextBox.Text;
```

If you do use dynamically generated queries, consider disabling query plan caching to avoid unnecessary memory consumption for cache entries that are unlikely to be reused.

Query plan caching on static queries and parameterized queries can provide performance benefits. The following is an example of a static query:

```
var query = "SELECT sp.SalesYTD FROM AdventureWorksEntities.SalesPerson as sp";
```

For queries to be matched properly by the query plan cache, they should comply with the following requirements:

- Query text should be a constant pattern, preferably a constant string or a resource.
- EntityParameter or ObjectParameter should be used wherever a user-supplied value must be passed.

You should avoid the following query patterns, which unnecessarily consume slots in the query plan cache:

- Changes to letter case in the text.
- Changes to white space.
- Changes to literal values.
- Changes to text inside comments.

See also

- Entity SQL Overview

Namespaces (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Entity SQL introduces namespaces to avoid name conflicts for global identifiers such as type names, entity sets, functions, and so on. The namespace support in Entity SQL is similar to the namespace support in the .NET Framework.

Entity SQL provides two forms of the USING clause: qualified namespaces (where a shorter alias is provided for the namespace), and unqualified namespaces, as illustrated in the following example:

```
USING System.Data;
```

```
USING tsql = System.Data;
```

Name Resolution Rules

If an identifier cannot be resolved in the local scopes, Entity SQL tries to locate the name in the global scopes (the namespaces). Entity SQL first tries to match the identifier (prefix) with one of the qualified namespaces. If there is a match, Entity SQL tries to resolve the rest of the identifier in the specified namespace. If no match is found, an exception is thrown.

Next, Entity SQL tries to search all unqualified namespaces (specified in the prolog) for the identifier. If the identifier can be located in exactly one namespace, that location is returned. If more than one namespace has a match for that identifier, an exception is thrown. If no namespace can be identified for the identifier, Entity SQL passes the name onto the next outward scope (the DbCommand or DbConnection object), as illustrated in the following example:

```
SELECT TREAT(p AS NamespaceName.Employee)
FROM ContainerName.Person AS p
WHERE p IS OF (NamespaceName.Employee)
```

Differences from the .NET Framework

In the .NET Framework, you can use partially qualified namespaces. Entity SQL does not allow this.

ADO.NET Usage

Queries are expressed through ADO.NET DbCommand objects. DbCommand objects can be built over DbConnection objects. Namespaces can also be specified as part of the DbCommand and DbConnection objects. If Entity SQL cannot resolve an identifier within the query itself, the external namespaces are probed (based on similar rules).

See also

- Entity SQL Reference
- Entity SQL Overview

Identifiers (Entity SQL)

11/8/2022 • 6 minutes to read • Edit Online

Identifiers are used in Entity SQL to represent query expression aliases, variable references, properties of objects, functions, and so on. Entity SQL provides two kinds of identifiers: simple identifiers and quoted identifiers.

Simple Identifiers

A simple identifier in Entity SQL is a sequence of alphanumeric and underscore characters. The first character of the identifier must be an alphabetical character (a-z or A-Z).

Quoted Identifiers

A quoted identifier is any sequence of characters enclosed in square brackets ([]). Quoted identifiers let you specify identifiers with characters that are not valid in identifiers. All characters between the square brackets become part of the identifier, including all white space.

A quoted identifier cannot include the following characters:

- Newline.
- Carriage returns.
- Tabs.
- Backspace.
- Additional square brackets (that is, square brackets within the square brackets that delineate the identifier).

A quoted-identifier can include Unicode characters.

Quoted identifiers enable you to create property name characters that are not valid in identifiers, as illustrated in the following example:

```
SELECT c.ContactName AS [Contact Name] FROM customers AS c
```

You can also use quoted identifiers to specify an identifier that is a reserved keyword of Entity SQL. For example, if the type `Email` has a property named "From", you can disambiguate it from the reserved keyword FROM by using square brackets, as follows:

```
SELECT e.[From] FROM emails AS e
```

You can use a quoted identifier on the right side of a dot (.) operator.

```
SELECT t FROM ts as t WHERE t.[property] == 2
```

To use the square bracket in an identifier, add an extra square bracket. In the following example "`abc]`" is the identifier:

```
SELECT t from ts as t WHERE t.[abc]]] == 2
```

For quoted identifier comparison semantics, see Input Character Set.

Aliasing Rules

We recommend specifying aliases in Entity SQL queries whenever needed, including the following Entity SQL constructs:

- Fields of a row constructor.
- Items in the FROM clause of a query expression.
- Items in the SELECT clause of a query expression.
- Items in the GROUP BY clause of a query expression.

Valid Aliases

Valid aliases in Entity SQL are any simple identifier or quoted identifier.

Alias Generation

If no alias is specified in an Entity SQL query expression, Entity SQL tries to generate an alias based on the following simple rules:

- If the query expression (for which the alias is unspecified) is a simple or quoted identifier, that identifier is used as the alias. For example, `ROW(a, [b])` becomes `ROW(a AS a, [b] AS [b])`.
- If the query expression is a more complex expression, but the last component of that query expression is a simple identifier, then that identifier is used as the alias. For example, `ROW(a.a1, b.[b1])` becomes `ROW(a.a1 AS a1, b.[b1] AS [b1])`.

We recommend that you do not use implicit aliasing if you want to use the alias name later. Anytime aliases (implicit or explicit) conflict or are repeated in the same scope, there will be a compile error. An implicit alias will pass compilation even if there is an explicit or implicit alias of the same name.

Implicit aliases are autogenerated based on user input. For example, the following line of code will generate NAME as an alias for both columns and therefore will conflict.

```
SELECT product.NAME, person.NAME
```

The following line of code, which uses explicit aliases, will also fail. However, the failure will be more apparent by reading the code.

```
SELECT 1 AS X, 2 AS X …
```

Scoping Rules

Entity SQL defines scoping rules that determine when particular variables are visible in the query language. Some expressions or statements introduce new names. The scoping rules determine where those names can be used, and when or where a new declaration with the same name as another can hide its predecessor.

When names are defined in an Entity SQL query, they are said to be defined within a scope. A scope covers an entire region of the query. All expressions or name references within a certain scope can see names that are defined within that scope. Before a scope begins and after it ends, names that are defined within the scope cannot be referenced.

Scopes can be nested. Parts of Entity SQL introduce new scopes that cover entire regions, and these regions can contain other Entity SQL expressions that also introduce scopes. When scopes are nested, references can be made to names that are defined in the innermost scope, which contains the reference. References can also be made to any names that are defined in any outer scopes. Any two scopes defined within the same scope are

considered sibling scopes. References cannot be made to names that are defined within sibling scopes.

If a name declared in an inner scope matches a name declared in an outer scope, references within the inner scope or within scopes declared within that scope refer only to the newly declared name. The name in the outer scope is hidden.

Even within the same scope, names cannot be referenced before they are defined.

Global names can exist as part of the execution environment. This can include names of persistent collections or environment variables. For a name to be global, it must be declared in the outermost scope.

Parameters are not in a scope. Because references to parameters include special syntax, names of parameters will never collide with other names in the query.

Query Expressions

An Entity SQL query expression introduces a new scope. Names that are defined in the FROM clause are introduced into the from scope in order of appearance, left to right. In the join list, expressions can refer to names that were defined earlier in the list. Public properties (fields and so on) of elements identified in the FROM clause are not added to the from-scope. They must be always referenced by the alias-qualified name. Typically, all parts of the SELECT expression are considered within the from-scope.

The GROUP BY clause also introduces a new sibling scope. Each group can have a group name that refers to the collection of elements in the group. Each grouping expression will also introduce a new name into the group-scope. Additionally, the nest aggregate (or the named group) is also added to the scope. The grouping expressions themselves are within the from-scope. However, when a GROUP BY clause is used, the select-list (projection), HAVING clause, and ORDER BY clause are considered to be within the group-scope, and not the from-scope. Aggregates receive special treatment, as described in the following bulleted list.

The following are additional notes about scopes:

- The select-list can introduce new names into the scope, in order. Projection expressions to the right might refer to names projected on the left.
- The ORDER BY clause can refer to names (aliases) specified in the select list.
- The order of evaluation of clauses within the SELECT expression determines the order that names are introduced into the scope. The FROM clause is evaluated first, followed by the WHERE clause, GROUP BY clause, HAVING clause, SELECT clause, and finally the ORDER BY clause.

Aggregate Handling

Entity SQL supports two forms of aggregates: collection-based aggregates and group-based aggregates. Collection-based aggregates are the preferred construct in Entity SQL, and group-based aggregates are supported for SQL compatibility.

When resolving an aggregate, Entity SQL first tries to treat it as a collection-based aggregate. If that fails, Entity SQL transforms the aggregate input into a reference to the nest aggregate and tries to resolve this new expression, as illustrated in the following example.

```
AVG(t.c) becomes AVG(group..(t.c))
```

See also

- Entity SQL Reference
- Entity SQL Overview
- Input Character Set

Parameters (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Parameters are variables that are defined outside Entity SQL, usually through a binding API that is used by a host language. Each parameter has a name and a type. Parameter names are defined in query expressions with the at (@) symbol as a prefix. This disambiguates them from the names of properties or other names that are defined in the query.

The host-language binding API provides APIs for binding parameters.

Example

```
SELECT c
      FROM LOB.Customers AS c
      WHERE c.Name = @name
```

See also

- Entity SQL Reference
- Entity SQL Overview

Variables (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Variable

A variable expression is a reference to a named expression defined in the current scope. A variable reference must be a valid Entity SQL identifier, as defined in Identifiers.

The following example shows the use of a variable in the expression. The `c` in the FROM clause is the definition of the variable. The use of `c` in the SELECT clause represents the variable reference.

```
select c
from LOB.customers as c
```

See also

- Identifiers
- Parameters
- Entity SQL Overview

Unsupported expressions

11/8/2022 • 2 minutes to read • Edit Online

This topic describes Transact-SQL expressions that are not supported in Entity SQL. For more information, see How Entity SQL Differs from Transact-SQL.

Quantified predicates

Transact-SQL allows constructs of the following form:

```
sal > all (select salary from employees)
sal > any (select salary from employees)
```

Entity SQL, however, does not support such constructs. Equivalent expressions can be written in Entity SQL as follows:

```
not exists(select 0 from employees as e where sal <= e.salary)
exists(select 0 from employees as e where sal > e.salary)
```

* operator

Transact-SQL supports the use of the * operator in the SELECT clause to indicate that all columns should be projected out. This is not supported in Entity SQL.

See also

- Entity SQL Overview
- How Entity SQL Differs from Transact-SQL

Literals (Entity SQL)

11/8/2022 • 5 minutes to read • Edit Online

This topic describes Entity SQL support for literals.

Null

The null literal is used to represent the value null for any type. A null literal is compatible with any type.

Typed nulls can be created by a cast over a null literal. For more information, see CAST.

For rules about where free floating null literals can be used, see Null Literals and Type Inference.

Boolean

Boolean literals are represented by the keywords `true` and `false`.

Integer

Integer literals can be of type Int32 or Int64. An Int32 literal is a series of numeric characters. An Int64 literal is series of numeric characters followed by an uppercase L.

Decimal

A fixed-point number (decimal) is a series of numeric characters, a dot (.) and another series of numeric characters followed by an uppercase "M".

Float, Double

A double-precision floating point number is a series of numeric characters, a dot (.) and another series of numeric characters possibly followed by an exponent. A single-precisions floating point number (or float) is a double-precision floating point number syntax followed by the lowercase f.

String

A string is a series of characters enclosed in quote marks. Quotes can be either both single-quotes (`'`) or both double-quotes ("). Character string literals can be either Unicode or non-Unicode. To declare a character string literal as Unicode, prefix the literal with an uppercase "N". The default is non-Unicode character string literals. There can be no spaces between the N and the string literal payload, and the N must be uppercase.

```
'hello' -- non-Unicode character string literal
N'hello' -- Unicode character string literal
"x"
N"This is a string!"
'so is THIS'
```

DateTime

A datetime literal is independent of locale and is composed of a date part and a time part. Both date and time parts are mandatory and there are no default values.

The date part must have the format: `YYYY` - `MM` - `DD`, where `YYYY` is a four digit year value between 0001 and 9999, `MM` is the month between 1 and 12 and `DD` is the day value that is valid for the given month `MM`.

The time part must have the format: `HH` : `MM` [: `SS` [.fffffff]], where `HH` is the hour value between 0 and 23, `MM` is the minute value between 0 and 59, `SS` is the second value between 0 and 59 and fffffff is the fractional second value between 0 and 9999999. All value ranges are inclusive. Fractional seconds are optional. Seconds are optional unless fractional seconds are specified; in this case, seconds are required. When seconds or fractional seconds are not specified, the default value of zero will be used instead.

There can be any number of spaces between the DATETIME symbol and the literal payload, but no new lines.

```
DATETIME'2006-10-1 23:11'
DATETIME'2006-12-25 01:01:00.0000000' -- same as DATETIME'2006-12-25 01:01'
```

Time

A time literal is independent of locale and composed of a time part only. The time part is mandatory and there is no default value. It must have the format HH:MM[:SS[.fffffff]], where HH is the hour value between 0 and 23, MM is the minute value between 0 and 59, SS is the second value between 0 and 59, and fffffff is the second fraction value between 0 and 9999999. All value ranges are inclusive. Fractional seconds are optional. Seconds are optional unless fractional seconds are specified; in this case, seconds are required. When seconds or fractions are not specified, the default value of zero will be used instead.

There can be any number of spaces between the TIME symbol and the literal payload, but no new lines.

```
TIME'23:11'
TIME'01:01:00.1234567'
```

DateTimeOffset

A datetimeoffset literal is independent of locale and composed of a date part, a time part, and an offset part. All date, time, and offset parts are mandatory and there are no default values. The date part must have the format YYYY-MM-DD, where YYYY is a four digit year value between 0001 and 9999, MM is the month between 1 and 12, and DD is the day value that is valid for the given month. The time part must have the format HH:MM[:SS[.fffffff]], where HH is the hour value between 0 and 23, MM is the minute value between 0 and 59, SS is the second value between 0 and 59, and fffffff is the fractional second value between 0 and 9999999. All value ranges are inclusive. Fractional seconds are optional. Seconds are optional unless fractional seconds are specified; in this case, seconds are required. When seconds or fractions are not specified, the default value of zero will be used instead. The offset part must have the format {+|-}HH:MM, where HH and MM have the same meaning as in the time part. The range of the offset, however, must be between -14:00 and + 14:00

There can be any number of spaces between the DATETIMEOFFSET symbol and the literal payload, but no new lines.

```
DATETIMEOFFSET'2006-10-1 23:11 +02:00'
DATETIMEOFFSET'2006-12-25 01:01:00.0000000 -08:30'
```

> **NOTE**
>
> A valid Entity SQL literal value can fall outside the supported ranges for CLR or the data source. This might result in an exception

Binary

A binary string literal is a sequence of hexadecimal digits delimited by single quotes following the keyword binary or the shortcut symbol `X` or `x`. The shortcut symbol `X` is case insensitive. A zero or more spaces are allowed between the keyword `binary` and the binary string value.

Hexadecimal characters are also case insensitive. If the literal is composed of an odd number of hexadecimal digits, the literal will be aligned to the next even hexadecimal digit by prefixing the literal with a hexadecimal zero digit. There is no formal limit on the size of the binary string.

```
Binary'00ffaabb'
X'ABCabc'
BINARY    '0f0f0f0F0F0F0F0F0F0F'
X'' -- empty binary string
```

Guid

A `GUID` literal represents a globally unique identifier. It is a sequence formed by the keyword `GUID` followed by hexadecimal digits in the form known as *registry* format: 8-4-4-4-12 enclosed in single quotes. Hexadecimal digits are case insensitive.

There can be any number of spaces between the GUID symbol and the literal payload, but no new lines.

```
Guid'1afc7f5c-ffa0-4741-81cf-f12eAAb822bf'
GUID  '1AFC7F5C-FFA0-4741-81CF-F12EAAB822BF'
```

See also

- Entity SQL Overview

Null Literals and Type Inference (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Null literals are compatible with any type in the Entity SQL type system. However, for the type of a null literal to be inferred correctly, Entity SQL imposes certain constraints on where a null literal can be used.

Typed Nulls

Typed nulls can be used anywhere. Type inference is not required for typed nulls because the type is known. For example, you can construct a null of type Int16 with the following Entity SQL construct:

```
(cast(null as Int16))
```

Free-Floating Null Literals

Free-floating null literals can be used in the following contexts:

- As an argument to a CAST or TREAT expression. This is the recommended way to produce a typed null expression.
- As an argument to a method or a function. Standard overload rules apply.
- As one of the arguments to an arithmetic expression such as +, -, or /. The other arguments cannot be null literals, otherwise type inference is not possible.
- As any of the arguments to a logical expression (AND, OR, or NOT). All the arguments are known to be of type Boolean.
- As the argument to an IS NULL or IS NOT NULL expression.
- As one or more of the arguments to a LIKE expression. All arguments are expected to be strings.
- As one or more of the arguments to a named-type constructor.
- As one or more of the arguments to a multiset constructor. At least one argument to the multiset constructor must be an expression that is not a null literal.
- As one or more of the THEN or ELSE expressions in a CASE expression. At least one of the THEN or ELSE expressions in the CASE expression must be an expression other than a null literal.

Free-floating null literals cannot be used in other scenarios. For example, they cannot be used as arguments to a row constructor.

See also

- Entity SQL Overview

Input Character Set (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Entity SQL accepts UNICODE characters encoded in UTF-16.

String literals can contain any UTF-16 character enclosed in single quotes. For example, N'文字列リテラル'. When string literals are compared, the original UTF-16 values are used. For example, N'ABC' is different in Japanese and Latin codepages.

Comments can contain any UTF-16 character.

Escaped identifiers can contain any UTF-16 character enclosed in square brackets. For example, [エスケープされた識別子]. The comparison of UTF-16 escaped identifiers is case insensitive. Entity SQL treats versions of letters that appear the same but are from different code pages as different characters. For example, [ABC] is equivalent to [abc] if the corresponding characters are from the same code page. However, if the same two identifiers are from different code pages, they are not equivalent.

White space is any UTF-16 white space character.

A newline is any normalized UTF-16 newline character. For example, '\n' and '\r\n' are considered newline characters, but '\r' is not a newline character.

Keywords, expressions, and punctuation can be any UTF-16 character that normalizes to Latin. For example, SELECT in a Japanese codepage is a valid keyword.

Keywords, expressions, and punctuation can only be Latin characters. `SELECT` in a Japanese codepage is not a keyword. +, -, *, /, =, (,), ', [,] and any other language construct not quoted here can only be Latin characters.

Simple identifiers can only be Latin characters. This avoids ambiguity during comparison, because original values are compared. For example, ABC would be different in Japanese and Latin codepages.

See also

- Entity SQL Overview

Query Expressions (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

A query expression combines many different query operators into a single syntax. Entity SQL provides various kinds of expressions, including the following: literals, parameters, variables, operators, functions, set operators, and so on. For more information, see Entity SQL Reference.

Clauses

A query expression is composed of a series of clauses that apply successive operations to a collection of objects. They are based on the same clauses found in standard a SQL select statement: SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BY.

Scope

Names defined in the FROM clause are introduced into the FROM scope in order of appearance, left to right. In the JOIN list, expressions can refer to names defined earlier in the list. Public properties of elements identified in the FROM clause are not added to the FROM scope: They must be always referenced through the alias-qualified name. Normally, all parts of the select expression are considered within the FROM scope.

See also

- Entity SQL Reference

Functions (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Entity SQL supports user-defined functions, canonical functions, and provider-specific functions. User-defined functions are specified in the conceptual model or inline in the query. For more information, see User-Defined Functions.

Canonical functions are predefined in the Entity Framework and should be supported by data providers. Entity SQL commands will fail if a user calls a function that is not supported by a provider. Therefore, canonical functions are generally recommended over store-specific functions, which are in a provider-specific namespace. For more information, see Canonical Functions.

The Microsoft SQL Client Managed Provider provides a set of provider-specific functions. For more information, see SqlClient for Entity Framework Functions.

In This Section

User-Defined Functions

Function Overload Resolution

Aggregate Functions

See also

- Entity SQL Overview

User-Defined Functions (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Entity SQL supports calling user-defined functions in a query. You can define these functions inline with the query (see How to: Call a User-Defined Function) or as part of the conceptual model (see How to: Define Custom Functions in the Conceptual Model). Conceptual model functions are defined as an Entity SQL command in the DefiningExpression element of a Function element in the conceptual model.

Entity SQL enables you to define functions in the query command itself. The FUNCTION operator defines inline functions. You can define multiple functions in a single command, and these functions can have the same function name, as long as the function signatures are unique. For more information, see Function Overload Resolution.

See also

- Functions

Function Overload Resolution (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

This topic describes how Entity SQL functions are resolved.

More than one function can be defined with the same name, as long as the functions have unique signatures.

When this is the case, the following criteria must be applied to determine which function is referenced by a given expression. These criteria are applied in sequence. The first criterion that applies only to a single function is the resolved function.

1. **Parameter number**. The function has the same number of parameters specified in the expression.
2. **Exact match on type**. Each argument type of the function exactly matches the parameter type, or is the null literal.
3. **Match on subtype**. Each argument type of the function exactly matches or is a sub-type of the parameter type, or the argument is the null literal. In the event that several functions differ only in the number of sub-type conversions required, the function with the least number of sub-type conversions is the resolved function.
4. **Match on subtype or type promotion**. Each argument type of the function exactly matches, is a sub-type of, or can be promoted to the parameter type, or the argument is the null literal. Again, in the event that several functions differ only in the number of sub-type conversions and promotions, the function with the least number of sub-type conversions and promotions is the resolved function.

If none of these criteria result in a single function being selected, the function invocation expression is ambiguous.

Even if a single function can be extracted using these rules, the arguments still might not match the parameters. An error is raised in this case.

For user-defined functions, the definition for an inline query function takes precedence even when a model-defined function exists with a signature that is a better match for the user-defined function.

See also

- Entity SQL Reference
- Entity SQL Overview
- Functions

Aggregate Functions (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

An aggregate is a language construct that condenses a collection into a scalar as a part of a group operation. Entity SQL aggregates come in two forms:

- Entity SQL collection functions that may be used anywhere in an expression. This includes using aggregate functions in projections and predicates that act on collections. Collection functions are the preferred mode of specifying aggregates in Entity SQL.
- Group aggregates in query expressions that have a GROUP BY clause. As in Transact-SQL, group aggregates accept DISTINCT and ALL as modifiers to the aggregate input.

Entity SQL first tries to interpret an expression as a collection function and if the expression is in the context of a SELECT expression it interprets it as a group aggregate.

Entity SQL defines a special aggregate operator called GROUPPARTITION. This operator enables you to get a reference to the grouped input set. This allows more advanced grouping queries, where the results of the GROUP BY clause can be used in places other than group aggregate or collection functions.

Collection Functions

Collection functions operate on collections and return a scalar value. For example, if `orders` is a collection of all `orders`, you can calculate the earliest ship date with the following expression:

```
min(select value o.ShipDate from LOB.Orders as o)
```

Group Aggregates

Group aggregates are calculated over a group result as defined by the GROUP BY clause. The GROUP BY clause partitions data into groups. For each group in the result, the aggregate function is applied and a separate aggregate is calculated by using the elements in each group as inputs to the aggregate calculation. When a GROUP BY clause is used in a SELECT expression, only grouping expression names, aggregates, or constant expressions may be present in the projection, HAVING, or ORDER BY clause.

The following example calculates the average quantity ordered for each product.

```
select p, avg(ol.Quantity) from LOB.OrderLines as ol
```

```
group by ol.Product as p
```

It is possible to have a group aggregate without an explicit GROUP BY clause in the SELECT expression. All elements will be treated as a single group, equivalent to the case of specifying a grouping based on a constant.

```
select avg(ol.Quantity) from LOB.OrderLines as ol
```

```
select avg(ol.Quantity) from LOB.OrderLines as ol group by 1
```

Expressions used in the GROUP BY clause are evaluated by using the same name-resolution scope that would be visible to the WHERE clause expression.

See also

- Functions

Operator Precedence (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

When an Entity SQL query has multiple operators, operator precedence determines the sequence in which the operations are performed. The order of execution can significantly affect the query result.

Operators have the precedence levels shown in the following table. An operator with a higher level is evaluated before an operator with a lower level.

LEVEL	OPERATION TYPE	OPERATOR
1	Primary	`. , [] ()`
2	Unary	`! not`
3	Multiplicative	`* / %`
4	Additive	`+ -`
5	Ordering	`< > <= >=`
6	Equality	`= != <>`
7	Conditional AND	`and &&`
8	Conditional OR	`or &#124;&#124;`

When two operators in an expression have the same operator precedence level, they are evaluated left to right, based on their position in the query. For example, `x+y-z` is evaluated as `(x+y)-z`.

You can use parentheses to override the defined precedence of the operators in a query. Everything within parentheses is evaluated first to yield a single result before that result can be used by any operator outside the parentheses. For example, `x+y*z` multiplies `y` by `z` and then adds `x`, but `(x+y)*z` adds `x` to `y` and then multiplies the result by `z`.

See also

- Entity SQL Overview

Paging (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Physical paging can be performed by using the SKIP and LIMIT sub-clauses in the ORDER BY clause. To perform physical paging deterministically, you should use SKIP and LIMIT. If you only want to restrict the number of rows in the result in a non-deterministic way, you should use TOP. TOP and SKIP/LIMIT are mutually exclusive.

TOP Overview

The SELECT clause can have an optional TOP sub-clause following the optional ALL/DISTINCT modifier. The TOP sub-clause specifies that only the first set of rows will be returned from the query result. For more information, see TOP.

SKIP And LIMIT Overview

SKIP and LIMIT are part of the ORDER BY clause. If a SKIP expression sub-clause is present in a ORDER BY clause, the results will be sorted according to the sort specification and the result set will include row(s) starting from the next row immediately after the SKIP expression. For example, SKIP 5 will skip the first five rows and return from the sixth row forward. If a LIMIT expression sub-clause is present in an ORDER BY clause, the query will be sorted according to the sort specification and the resulting number of rows will be restricted by the LIMIT expression. For instance, LIMIT 5 will restrict the result set to five instances or rows. SKIP and LIMIT do not have to be used together; you can use just SKIP or just LIMIT with ORDER BY clause. For more information, see the following topics:

- SKIP
- LIMIT
- ORDER BY

See also

- Entity SQL Reference
- Entity SQL Overview
- How to: Page Through Query Results

Comparison Semantics (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Performing any of the following Entity SQL operators involves comparison of type instances:

Explicit comparison

Equality operations:

- =
- !=

Ordering operations:

- <
- <=
- >
- >=

Nullability operations:

- IS NULL
- IS NOT NULL

Explicit distinction

Equality distinction:

- DISTINCT
- GROUP BY

Ordering distinction:

- ORDER BY

Implicit distinction

Set operations and predicates (equality):

- UNION
- INTERSECT
- EXCEPT
- SET
- OVERLAPS

Item predicates (equality):

- IN

Supported Combinations

The following table shows all the supported combinations of comparison operators for each kind of type:

TYPE	= !=	GROUP BY DISTINCT	UNION INTERSECT EXCEPT SET OVERLAPS	IN	< <= > >=	ORDER BY	IS NULL IS NOT NULL
Entity type	Ref[1]	All properties[2]	All properties[2]	All properties[2]	Throw[3]	Throw[3]	Ref[1]
Complex type	Throw[3]	Throw[3]	Throw[3]	Throw[3]	Throw[3]	Throw[3]	Throw[3]
Row	All properties[4]	All properties[4]	All properties[4]	Throw[3]	Throw[3]	All properties[4]	Throw[3]
Primitive type	Provider-specific	Provider-specific	Provider-specific	Provider-specific	Provider-specific	Provider-specific	Provider-specific
Multiset	Throw[3]	Throw[3]	Throw[3]	Throw[3]	Throw[3]	Throw[3]	Throw[3]
Ref	Yes[5]	Yes[5]	Yes[5]	Yes[5]	Throw	Throw	Yes[5]
Association type	Throw[3]	Throw	Throw	Throw	Throw[3]	Throw[3]	Throw[3]

[1]The references of the given entity type instances are implicitly compared, as shown in the following example:

```
SELECT p1, p2
FROM AdventureWorksEntities.Product AS p1
     JOIN AdventureWorksEntities.Product AS p2
WHERE p1 != p2 OR p1 IS NULL
```

An entity instance cannot be compared to an explicit reference. If this is attempted, an exception is thrown. For example, the following query will throw an exception:

```
SELECT p1, p2
FROM AdventureWorksEntities.Product AS p1
     JOIN AdventureWorksEntities.Product AS p2
WHERE p1 != REF(p2)
```

[2]Properties of complex types are flattened out before being sent to the store, so they become comparable (as long as all their properties are comparable). Also see [4].

[3]The Entity Framework runtime detects the unsupported case and throws a meaningful exception without engaging the provider/store.

[4]An attempt is made to compare all properties. If there is a property that is of a non-comparable type, such as text, ntext, or image, a server exception might be thrown.

[5]All individual elements of the references are compared (this includes the entity set name and all the key

properties of the entity type).

See also

- Entity SQL Overview

Composing Nested Entity SQL Queries

11/8/2022 • 2 minutes to read • Edit Online

Entity SQL is a rich functional language. The building block of Entity SQL is an expression. Unlike conventional SQL, Entity SQL is not limited to a tabular result set: Entity SQL supports composing complex expressions that can have literals, parameters, or nested expressions. A value in the expression can be parameterized or composed of some other expression.

Nested Expressions

A nested expression can be placed anywhere a value of the type it returns is accepted. For example:

```
-- Returns a hierarchical collection of three elements at top-level.
-- x must be passed in the parameter collection.
ROW(@x, {@x}, {@x, 4, 5}, {@x, 7, 8, 9})

-- Returns a hierarchical collection of one element at top-level.
-- x must be passed in the parameter collection.
{{{@x}}};
```

A nested query can be placed in a projection clause. For example:

```
-- Returns a collection of rows where each row contains an Address entity.
-- and a collection of references to its corresponding SalesOrderHeader entities.
SELECT address, (SELECT DEREF(soh)
                   FROM NAVIGATE(address, AdventureWorksModel.FK_SalesOrderHeader_Address_BillToAddressID)
AS soh)
                   AS salesOrderHeader FROM AdventureWorksEntities.Address AS address
```

In Entity SQL, nested queries must always be enclosed in parentheses:

```
-- Pseudo-Entity SQL
( SELECT …
FROM … )
UNION ALL
( SELECT …
FROM … );
```

The following example demonstrates how to properly nest expressions in Entity SQL: How to: Order the Union of Two Queries.

Nested Queries in Projection

Nested queries in the project clause might get translated into Cartesian product queries on the server. In some backend servers, including SQL Server, this can cause the TempDB table to get very large, which can adversely affect server performance.

The following is an example of such a query:

```
SELECT c, (SELECT c, (SELECT c FROM AdventureWorksModel.Vendor AS c  ) As Inner2 FROM
AdventureWorksModel.JobCandidate AS c  ) As Inner1 FROM AdventureWorksModel.EmployeeDepartmentHistory AS c
```

Ordering Nested Queries

In the Entity Framework, a nested expression can be placed anywhere in the query. Because Entity SQL allows great flexibility in writing queries, it is possible to write a query that contains an ordering of nested queries. However, the order of a nested query is not preserved.

```
-- The following query will order the results by last name.
SELECT C1.FirstName, C1.LastName
        FROM AdventureWorksModel.Contact as C1
        ORDER BY C1.LastName
```

```
-- In the following query, ordering of the nested query is ignored.
SELECT C2.FirstName, C2.LastName
    FROM (SELECT C1.FirstName, C1.LastName
        FROM AdventureWorksModel.Contact as C1
        ORDER BY C1.LastName) as C2
```

See also

- Entity SQL Overview

Nullable Structured Types (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

A `null` instance of a structured type is an instance that does not exist. This is different from an existing instance in which all properties have `null` values.

This topic describes the nullable structured types, including which types are nullable and which code patterns produce `null` instances of structured nullable types.

Kinds of Nullable Structured Types

There are three kinds of nullable structure types:

- Row types.
- Complex types.
- Entity types.

Code Patterns that Produce Null Instances of Structured Types

The following scenarios produce `null` instances:

- Shaping `null` as a structured type:

```
TREAT (NULL AS StructuredType)
```

- Upcasting of a base type to a derived type:

```
TREAT (BaseType AS DerivedType)
```

- Outer join on false condition:

```
Collection1 LEFT OUTER JOIN Collection2
ON FalseCondition
```

--or

```
Collection1 RIGHT OUTER JOIN Collection2
ON FalseCondition
```

--or

```
Collection1 FULL OUTER JOIN Collection2
ON FalseCondition
```

- Dereferencing a `null` reference:

```
DEREF(NullRef)
```

- Obtaining ANYELEMENT from an empty collection:

```
ANYELEMENT(EmptyCollection)
```

- Checking for `null` instances of structured types:

```
...
for (int i = 0; i < reader.FieldCount; i++)
{
    if (reader.IsDBNull(i))
    {
        Console.WriteLine("[NULL]");
    }
    else
    {
        Console.WriteLine(reader.GetValue(i).ToString());
    }
}
```

See also

- Entity SQL Overview

Entity SQL reference

11/8/2022 • 5 minutes to read • Edit Online

This section contains Entity SQL reference articles. This article summarizes and groups the Entity SQL operators by category.

Arithmetic operators

Arithmetic operators perform mathematical operations on two expressions of one or more numeric data types. The following table lists the Entity SQL arithmetic operators:

OPERATOR	USE
+ (Add)	Addition.
/ (Divide)	Division.
% (Modulo)	Returns the remainder of a division.
* (Multiply)	Multiplication.
- (Negative)	Negation.
- (Subtract)	Subtraction.

Canonical functions

Canonical functions are supported by all data providers and can be used by all querying technologies. The following table lists the canonical functions:

FUNCTION	TYPE
Aggregate Entity SQL Canonical Functions	Discusses aggregate Entity SQL canonical functions.
Math Canonical Functions	Discusses math Entity SQL canonical functions.
String Canonical Functions	Discusses string Entity SQL canonical functions.
Date and Time Canonical Functions	Discusses date and time Entity SQL canonical functions.
Bitwise Canonical Functions	Discusses bitwise Entity SQL canonical functions.
Other Canonical Functions	Discusses functions not classified as bitwise, date/time, string, math, or aggregate.

Comparison operators

Comparison operators are defined for the following types: `Byte`, `Int16`, `Int32`, `Int64`, `Double`, `Single`, `Decimal`, `String`, `DateTime`, `Date`, `Time`, `DateTimeOffset`. Implicit type promotion occurs for the operands

before the comparison operator is applied. Comparison operators always yield Boolean values. When at least one of the operands is `null`, the result is `null`.

Equality and inequality are defined for any object type that has identity, such as the `Boolean` type. Non-primitive objects with identity are considered equal if they share the same identity. The following table lists the Entity SQL comparison operators:

OPERATOR	DESCRIPTION
= (Equals)	Compares the equality of two expressions.
> (Greater Than)	Compares two expressions to determine whether the left expression has a value greater than the right expression.
>= (Greater Than or Equal To)	Compares two expressions to determine whether the left expression has a value greater than or equal to the right expression.
IS [NOT] NULL	Determines if a query expression is null.
< (Less Than)	Compares two expressions to determine whether the left expression has a value less than the right expression.
<= (Less Than or Equal To)	Compares two expressions to determine whether the left expression has a value less than or equal to the right expression.
[NOT] BETWEEN	Determines whether an expression results in a value in a specified range.
!= (Not Equal To)	Compares two expressions to determine whether the left expression isn't equal to the right expression.
[NOT] LIKE	Determines whether a specific character string matches a specified pattern.

Logical and case expression operators

Logical operators test for the truth of a condition. The CASE expression evaluates a set of Boolean expressions to determine the result. The following table lists the logical and CASE expression operators:

OPERATOR	DESCRIPTION
&& (Logical AND)	Logical AND.
! (Logical NOT)	Logical NOT.
|| (Logical OR)	Logical OR.
CASE	Evaluates a set of Boolean expressions to determine the result.
THEN	The result of a WHEN clause when it evaluates to true.

Query operators

Query operators are used to define query expressions that return entity data. The following table lists query operators:

OPERATOR	USE
FROM	Specifies the collection that is used in SELECT statements.
GROUP BY	Specifies groups into which objects that are returned by a query (SELECT) expression are to be placed.
GroupPartition	Returns a collection of argument values, projected off the group partition to which the aggregate is related.
HAVING	Specifies a search condition for a group or an aggregate.
LIMIT	Used with the ORDER BY clause to performed physical paging.
ORDER BY	Specifies the sort order that is used on objects returned in a SELECT statement.
SELECT	Specifies the elements in the projection that are returned by a query.
SKIP	Used with the ORDER BY clause to performed physical paging.
TOP	Specifies that only the first set of rows will be returned from the query result.
WHERE	Conditionally filters data that is returned by a query.

Reference operators

A reference is a logical pointer (foreign key) to a specific entity in a specific entity set. Entity SQL supports the following operators to construct, deconstruct, and navigate through references:

OPERATOR	USE
CREATEREF	Creates references to an entity in an entity set.
DEREF	Dereferences a reference value and produces the result of that dereference.
KEY	Extracts the key of a reference or of an entity expression.
NAVIGATE	Allows you to navigate over the relationship from one entity type to another
REF	Returns a reference to an entity instance.

Set operators

Entity SQL provides various powerful set operations. This includes set operators similar to Transact-SQL operators such as UNION, INTERSECT, EXCEPT, and EXISTS. Entity SQL also supports operators for duplicate elimination (SET), membership testing (IN), and joins (JOIN). The following table lists the Entity SQL set operators:

OPERATOR	USE
ANYELEMENT	Extracts an element from a multivalued collection.
EXCEPT	Returns a collection of any distinct values from the query expression to the left of the EXCEPT operand that aren't also returned from the query expression to the right of the EXCEPT operand.
[NOT] EXISTS	Determines if a collection is empty.
FLATTEN	Converts a collection of collections into a flattened collection.
[NOT] IN	Determines whether a value matches any value in a collection.
INTERSECT	Returns a collection of any distinct values that are returned by both the query expressions on the left and right sides of the INTERSECT operand.
OVERLAPS	Determines whether two collections have common elements.
SET	Used to convert a collection of objects into a set by yielding a new collection with all duplicate elements removed.
UNION	Combines the results of two or more queries into a single collection.

Type operators

Entity SQL provides operations that allow the type of an expression (value) to be constructed, queried, and manipulated. The following table lists operators that are used to work with types:

OPERATOR	USE
CAST	Converts an expression of one data type to another.
COLLECTION	Used in a FUNCTION operation to declare a collection of entity types or complex types.
IS [NOT] OF	Determines whether the type of an expression is of the specified type or one of its subtypes.
OFTYPE	Returns a collection of objects from a query expression that is of a specific type.
Named Type Constructor	Used to create instances of entity types or complex types.
MULTISET	Creates an instance of a multiset from a list of values.

OPERATOR	USE
ROW	Constructs anonymous, structurally typed records from one or more values.
TREAT	Treats an object of a particular base type as an object of the specified derived type.

Other operators

The following table lists other Entity SQL operators:

OPERATOR	USE
+ (String Concatenation)	Used to concatenate strings in Entity SQL.
. (Member Access)	Used to access the value of a property or field of an instance of structural conceptual model type.
-- (Comment)	Include Entity SQL comments.
FUNCTION	Defines an inline function that can be executed in an Entity SQL query.

See also

- Entity SQL Language

+ (Add)

11/8/2022 • 2 minutes to read • Edit Online

Adds two numbers.

Syntax

```
expression + expression
```

Arguments

`expression`
Any valid expression of any one of the numeric data types.

Result Types

The data type that results from the implicit type promotion of the two arguments. For more information about implicit type promotion, see Type System.

Remarks

For EDM.String types, addition is concatenation.

Example

The following Entity SQL query uses the + arithmetic operator to add two numbers. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE product FROM AdventureWorksEntities.Products AS product
    where product.ListPrice =  @price1 + @price2
```

See also

- Entity SQL Reference
- Conceptual Model Types (CSDL)

+ (String Concatenation) (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Concatenates two strings.

Syntax

```
expression + expression
```

Arguments

`expression`
Any valid expression of the EDM.String data types. Both expressions must be of the same data type, or one expression must be able to be implicitly converted to the data type of the other expression.

Result Types

The data type that results from the implicit type promotion of the two arguments. For more information about implicit type promotion, see Type System.

Example

The following Entity SQL query uses the + operator to concatenates two strings. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns PrimitiveType Results.
2. Pass the following query as an argument to the `ExecutePrimitiveTypeQuery` method:

```
SELECT VALUE 'Name=[' + e.Name + ']' FROM
    AdventureWorksEntities.Products AS e
```

See also

- Entity SQL Reference
- Conceptual Model Types (CSDL)

- (Negative) (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Returns the negative of the value of a numeric expression.

Syntax

```
- expression
```

Arguments

`expression`
Any valid expression of any one of the numeric data types.

Result Types

The data type of `expression`.

Remarks

If `expression` is an unsigned type, the result type will be the signed type that most closely relates to the type of `expression`. For example, if `expression` is of type Byte, a value of type Int16 will be returned.

Example

The following Entity SQL query uses the - arithmetic operator to return the negative of the value of a numeric expression. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE product FROM AdventureWorksEntities.Products
    AS product WHERE product.ListPrice = -(-@price)
```

See also

- Entity SQL Reference

- (Subtract) (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Subtracts two numbers.

Syntax

```
expression - expression
```

Arguments

`expression`
Any valid expression of any one of the numeric data types.

Result Types

The data type that results from the implicit type promotion of the two arguments. For more information about implicit type promotion, see Type System.

Example

The following Entity SQL query uses the - arithmetic operator to subtract two numbers. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE product FROM AdventureWorksEntities.Products
    AS product WHERE product.ListPrice = @price1 - @price2
```

See also

- Entity SQL Reference

* (Multiply) (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Multiplies two expressions.

Syntax

```
expression * expression
```

Arguments

`expression`
Any valid expression of any one of the numeric data types.

Result Types

The data type that results from the implicit type promotion of the two arguments. For more information about implicit type promotion, see Type System.

Example

The following Entity SQL query uses the * arithmetic operator to multiply two numbers. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE product FROM AdventureWorksEntities.Products
    AS product WHERE product.ListPrice = @price1 * @price2
```

See also

- Entity SQL Reference

/ (Divide) (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Divides one number by another.

Syntax

```
dividend / divisor
```

Arguments

`dividend`
The numeric expression to divide. `dividend` is any valid expression of any one of the numeric data types.

`divisor`
The numeric expression to divide the dividend by. `divisor` is any valid expression of any one of the numeric data types.

Result Types

The data type that results from the implicit type promotion of the two arguments. For more information about implicit type promotion, see Type System.

Example

The following Entity SQL query uses the / arithmetic operator to divide one number by another. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE product FROM AdventureWorksEntities.Products
    AS product WHERE product.ListPrice =  @price1 / @price2
```

See also

- Entity SQL Reference

% (Modulo) (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Returns the remainder of one expression divided by another.

Syntax

```
dividend % divisor
```

Arguments

`dividend`
The numeric expression to divide. `dividend` is any valid expression of any one of the numeric data types.

`divisor`
The numeric expression to divide the dividend by. `divisor` is any valid expression of any one of the numeric data types.

Result Types

Edm.Int32

Example

The following Entity SQL query uses the % arithmetic operator to return the remainder of one expression divided by another. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE product FROM AdventureWorksEntities.Products
  AS product WHERE product.ListPrice = @price1 % @price2
```

See also

- Entity SQL Reference

&& (AND) (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Returns `true` if both expressions are `true`; otherwise, `false` or `NULL`.

Syntax

```
boolean_expression AND boolean_expression
```

or

```
boolean_expression && boolean_expression
```

Arguments

`boolean_expression`
Any valid expression that returns a Boolean.

Remarks

Double ampersands (&&) have the same functionality as the `AND` operator.

The following matrix shows possible input value combinations and return values.

	`TRUE`	`FALSE`	`NULL`
`TRUE`	TRUE	FALSE	NULL
`FALSE`	FALSE	FALSE	FALSE
`NULL`	NULL	FALSE	NULL

Example

The following Entity SQL query demonstrates how to use the AND operator. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
-- AND
SELECT VALUE product FROM AdventureWorksEntities.Products
    AS product where product.ListPrice >  @price1 AND product.ListPrice <  @price2
-- &&
SELECT VALUE product FROM AdventureWorksEntities.Products
    AS product where product.ListPrice > @price1 && product.ListPrice < @price2
```

See also

- Entity SQL Reference

|| (OR) (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Combines two `Boolean` expressions.

Syntax

```
boolean_expression OR boolean_expression
-- or
boolean_expression || boolean_expression
```

Arguments

`boolean_expression` Any valid expression that returns a `Boolean`.

Return Value

`true` when either of the conditions is `true`; otherwise, `false`.

Remarks

OR is an Entity SQL logical operator. It is used to combine two conditions. When more than one logical operator is used in a statement, OR operators are evaluated after AND operators. However, you can change the order of evaluation by using parentheses.

Double vertical bars (||) have the same functionality as the OR operator.

The following matrix shows possible input value combinations and return values.

	`TRUE`	`FALSE`	`NULL`
`TRUE`	TRUE	TRUE	TRUE
`FALSE`	TRUE	FALSE	NULL
`NULL`	TRUE	NULL	NULL

Example

The following Entity SQL query uses the OR operator to combine two `Boolean` expressions. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
-- OR
SELECT VALUE product FROM AdventureWorksEntities.Products
    AS product
WHERE product.ListPrice = @price1 OR product.ListPrice = @price2
-- ||
SELECT VALUE product FROM AdventureWorksEntities.Products
    AS product
WHERE product.ListPrice = @price1 || product.ListPrice = @price2
```

See also

- Entity SQL Reference

! (NOT) (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Negates a `Boolean` expression.

Syntax

```
NOT boolean_expression
-- or
! boolean_expression
```

Arguments

`boolean_expression`
Any valid expression that returns a Boolean.

Remarks

The exclamation point (!) has the same functionality as the NOT operator.

Example

The following Entity SQL query uses the NOT operator to negates a `Boolean` expression. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
-- NOT
SELECT VALUE product FROM AdventureWorksEntities.Products
AS product WHERE product.ListPrice > @price1 AND NOT (product.ListPrice = @price2)
-- !
SELECT VALUE product FROM AdventureWorksEntities.Products
AS product WHERE product.ListPrice > @price1 AND ! (product.ListPrice = @price2)
```

See also

- Entity SQL Reference

= (Equals) (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Compares the equality of two expressions.

Syntax

```
expression = expression
-- or
expression == expression
```

Arguments

`expression`
Any valid expression. Both expressions must have implicitly convertible data types.

Result Types

`true` if the left expression is equal to the right expression; otherwise, `false`.

Remarks

The == operator is equivalent to =.

Example

The following Entity SQL query uses = comparison operator to compare the equality of two expressions. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE product FROM AdventureWorksEntities.Products
    AS product WHERE product.ListPrice = @price
```

See also

- Entity SQL Reference

> (Greater Than) (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Compares two expressions to determine whether the left expression has a value greater than the right expression.

Syntax

```
expression > expression
```

Arguments

`expression`
Any valid expression. Both expressions must have implicitly convertible data types.

Result Types

`true` if the left expression has a value greater than the right expression; otherwise, `false`.

Example

The following Entity SQL query uses > comparison operator to compare two expressions to determine whether the left expression has a value greater than the right expression. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE product FROM AdventureWorksEntities.Products
    AS product WHERE product.ListPrice > @price
```

See also

- Entity SQL Reference

>= (Greater Than or Equal To) (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Compares two expressions to determine whether the left expression has a value greater than or equal to the right expression.

Syntax

```
expression >= expression
```

Arguments

`expression`
Any valid expression. Both expressions must have implicitly convertible data types.

Result Types

`true` if the left expression has a value greater than or equal to the right expression; otherwise, `false`.

Example

The following Entity SQL query uses >= comparison operator to compare two expressions to determine whether the left expression has a value greater than or equal to the right expression. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE product FROM AdventureWorksEntities.Products
    AS product WHERE product.ListPrice >= @price
```

See also

- Entity SQL Reference

< (Less Than) (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Compares two expressions to determine whether the left expression has a value less than the right expression.

Syntax

```
expression < expression
```

Arguments

`expression`
Any valid expression. Both expressions must have implicitly convertible data types.

Result Types

`true` if the left expression has a value less than the right expression; otherwise, `false`.

Example

The following Entity SQL query uses < comparison operator to compare two expressions to determine whether the left expression has a value less than the right expression. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE product FROM AdventureWorksEntities.Products
    AS product WHERE product.ListPrice < @price
```

See also

- Entity SQL Reference

<= (Less Than or Equal To) (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Compares two expressions to determine whether the left expression has a value less than or equal to the right expression.

Syntax

```
expression <= expression
```

Arguments

`expression`
Any valid expression. Both expressions must have implicitly convertible data types.

Result Types

`true` if the left expression has a value less than or equal to the right expression; otherwise, `false`.

Example

The following Entity SQL query uses <= comparison operator to compare two expressions to determine whether the left expression has a value less than or equal to the right expression. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE product FROM AdventureWorksEntities.Products
    AS product WHERE product.ListPrice <= @price
```

See also

- Entity SQL Reference

!= (Not Equal To) (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Compares two expressions to determine whether the left expression is not equal to the right expression. The != (Not Equal To) operator is functionally equivalent to the <> operator.

Syntax

```
expression != expression
-- or
expression <> expression
```

Arguments

`expression`
Any valid expression. Both expressions must have implicitly convertible data types.

Result Types

`true` if the left expression is not equal to the right expression; otherwise, `false`.

Example

The following Entity SQL query uses the != operator to compare two expressions to determine whether the left expression is not equal to the right expression. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
-- !=
SELECT VALUE product FROM AdventureWorksEntities.Products
    AS product WHERE product.ListPrice != @price
-- <>
SELECT VALUE product FROM AdventureWorksEntities.Products
    AS product WHERE product.ListPrice <> @price
```

See also

- Entity SQL Reference

. (Member Access) (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

The dot operator (.) is the Entity SQL member access operator. You use the member access operator to yield the value of a property or field of an instance of structural conceptual model type.

Syntax

```
expression.identifier
```

Arguments

`expression` An instance of a structural conceptual model type.

`identifier` A property or field that belongs to an object instance.

Remarks

The dot (.) operator may be used to extract fields from a record, similar to extracting properties of a complex or entity type. For example, if n of type Name is a member of type Person, and p is an instance of type Person, then p.n is a legal member access expression that yields a value of type Name.

```
select p.Name.FirstName from LOB.Person as p
```

See also

- Entity SQL Reference

-- (Comment) (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Entity SQL queries can contain comments. Two dashes (`--`) start a comment line.

Syntax

```
-- text_of_comment
```

Arguments

`text_of_comment` Is the character string that contains the text of the comment.

Example

The following Entity SQL query demonstrates how to use comments. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE product FROM AdventureWorksEntities.Products AS product -- add a comment here
```

See also

- Entity SQL Overview
- Entity SQL Reference

ANYELEMENT (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Extracts an element from a multivalued collection.

Syntax

```
ANYELEMENT ( expression )
```

Arguments

`expression` Any valid query expression that returns a collection to extract an element from.

Return Value

A single element in the collection or an arbitrary element if the collection has more than one; if the collection is empty, returns `null`. If `collection` is a collection of type `Collection<T>`, then `ANYELEMENT(collection)` is a valid expression that yields an instance of type `T`.

Remarks

ANYELEMENT extracts an arbitrary element from a multivalued collection. For example, the following example attempts to extract a singleton element from the set `Customers`.

```
ANYELEMENT(Customers)
```

Example

The following Entity SQL query uses the ANYELEMENT operator to extract an element from a multivalued collection. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
ANYELEMENT((SELECT VALUE product from AdventureWorksEntities.Products as
                        product where product.ListPrice = @price))
```

See also

- Entity SQL Reference
- Nullable Structured Types

BETWEEN (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Determines whether an expression results in a value in a specified range. The Entity SQL BETWEEN expression has the same functionality as the Transact-SQL BETWEEN expression.

Syntax

```
expression [ NOT ] BETWEEN begin_expression AND end_expression
```

Arguments

`expression` Any valid expression to test for in the range defined by `begin_expression` and `end_expression`. `expression` must be the same type as both `begin_expression` and `end_expression`.

`begin_expression` Any valid expression. `begin_expression` must be the same type as both `expression` and `end_expression`. `begin_expression` should be less than `end_expression`, else the return value will be negated.

`end_expression` Any valid expression. `end_expression` must be the same type as both `expression` and `begin_expression`.

NOT Specifies that the result of BETWEEN be negated.

AND Acts as a placeholder that indicates `expression` should be within the range indicated by `begin_expression` and `end_expression`.

Return Value

`true` if `expression` is between the range indicated by `begin_expression` and `end_expression`; otherwise, `false`. `null` will be returned if `expression` is `null` or if `begin_expression` or `end_expression` is `null`.

Remarks

To specify an exclusive range, use the greater than (>) and less than (<) operators instead of BETWEEN.

Example

The following Entity SQL query uses BETWEEN operator to determine whether an expression results in a value in a specified range. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE product FROM AdventureWorksEntities.Products
    AS product where product.ListPrice BETWEEN @price1 AND @price2
```

See also

- Entity SQL Reference

CASE (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Evaluates a set of `Boolean` expressions to determine the result.

Syntax

```
CASE
     WHEN Boolean_expression THEN result_expression
    [ ...n ]
     [
    ELSE else_result_expression
     ]
END
```

Arguments

`n` Is a placeholder that indicates that multiple WHEN `Boolean_expression` THEN `result_expression` clauses can be used.

THEN `result_expression` Is the expression returned when `Boolean_expression` evaluates to `true`. `result expression` is any valid expression.

ELSE `else_result_expression` Is the expression returned if no comparison operation evaluates to `true`. If this argument is omitted and no comparison operation evaluates to `true`, CASE returns null. `else_result_expression` is any valid expression. The data types of `else_result_expression` and any `result_expression` must be the same or must be an implicit conversion.

WHEN `Boolean_expression` Is the `Boolean` expression evaluated when the searched CASE format is used. `Boolean_expression` is any valid `Boolean` expression.

Return Value

Returns the highest precedence type from the set of types in the `result_expression` and the optional `else_result_expression`.

Remarks

The Entity SQL case expression resembles the Transact-SQL case expression. You use the case expression to make a series of conditional tests to determine which expression will yield the appropriate result. This form of the case expression applies to a series of one or more `Boolean` expressions to determine the correct resulting expression.

The CASE function evaluates `Boolean_expression` for each WHEN clause in the order specified, and returns `result_expression` of the first `Boolean_expression` that evaluates to `true`. The remaining expressions are not evaluated. If no `Boolean_expression` evaluates to `true`, the Database Engine returns the `else_result_expression` if an ELSE clause is specified, or a null value if no ELSE clause is specified.

A CASE statement cannot return a multiset.

Example

The following Entity SQL query uses the CASE expression to evaluate a set of `Boolean` expressions in order to determine the result. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns PrimitiveType Results.
2. Pass the following query as an argument to the `ExecutePrimitiveTypeQuery` method:

```
CASE WHEN AVG({@score1,@score2,@score3}) < @total THEN TRUE ELSE FALSE END
```

See also

- THEN
- SELECT
- Entity SQL Reference

CAST (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Converts an expression of one data type to another.

Syntax

```
CAST ( expression AS data_type )
```

Arguments

`expression` Any valid expression that is convertible to `data_type`.

`data_type` The target system-supplied data type. It must be a primitive (scalar) type. The `data_type` used depends on the query space. If a query is executed with the EntityCommand, the data type is a type defined in the conceptual model. For more information, see CSDL Specification. If a query is executed with ObjectQuery<T>, the data type is a common language runtime (CLR) type.

Return Value

Returns the same value as `data_type`.

Remarks

The cast expression has similar semantics to the Transact-SQL CONVERT expression. The cast expression is used to convert a value of one type into a value of another type.

```
CAST( e as T )
```

If e is of some type S, and S is convertible to T, then the above expression is a valid cast expression. T must be a primitive (scalar) type.

Values for the precision and scale facets may optionally be provided when casting to `Edm.Decimal`. If not explicitly provided, the default values for precision and scale are 18 and 0, respectively. Specifically, the following overloads are supported for `Decimal`:

- `CAST( d as Edm.Decimal );`
- `CAST( d as Edm.Decimal(precision) );`
- `CAST( d as Edm.Decimal(precision, scale) );`

The use of a cast expression is considered an explicit conversion. Explicit conversions might truncate data or lose precision.

> **NOTE**
> CAST is only supported over primitive types and enumeration member types.

Example

The following Entity SQL query uses the CAST operator to cast an expression of one data type to another. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns PrimitiveType Results.
2. Pass the following query as an argument to the `ExecutePrimitiveTypeQuery` method:

```
SELECT VALUE cast(p.ListPrice as Edm.Int32)
    FROM AdventureWorksEntities.Products as p order by p.ListPrice
```

See also

- Entity SQL Reference

COLLECTION (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

The COLLECTION keyword is only used in the definition of an inline function. Collection functions are functions that operate on a collection of values and produce a scalar output.

Syntax

```
COLLECTION(type_definition)
```

Arguments

`type_definition`

An expression that returns a collection of supported types, rows, or references.

Remarks

For more information about the COLLECTION keyword, see Type Definitions.

Example

The following sample shows how to use the COLLECTION keyword to declare a collection of decimals as an argument for an inline query function.

```
USING Microsoft.Samples.Entity
Function MyAvg(dues Collection(Decimal)) AS
(
        Avg(select value due from dues as due where due > @price)
)
SELECT TOP(10) contactID, MyAvg(GroupPartition(order.TotalDue))
FROM AdventureWorksEntities.SalesOrderHeaders  AS order
GROUP BY order.Contact.ContactID as contactID;
```

See also

- Entity SQL Reference

CREATEREF (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Fabricates references to an entity in an entityset.

Syntax

```
CreateRef(entityset_identifier, row_typed_expression)
```

Arguments

`entityset_identifier`
The entityset identifier, not a string literal.

`row_typed_expression`
A row-typed expression that corresponds to the key properties of the entity type.

Remarks

`row_typed_expression` must be structurally equivalent to the key type for the entity. That is, it must have the same number and types of fields in the same order as the entity keys.

In the example below, Orders and BadOrders are both entitysets of type Order, and Id is assumed to be the single key property of Order. The example illustrates how we may produce a reference to an entity in BadOrders. Note that the reference may be dangling. That is, the reference may not actually identify a specific entity. In those cases, a `DEREF` operation on that reference returns a null.

```
SELECT CreateRef(LOB.BadOrders, row(o.Id))
FROM LOB.Orders AS o
```

Example

The following Entity SQL query uses the CREATEREF operator to fabricate references to an entity in an entity set. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE Key(CreateRef(AdventureWorksEntities.Products,
    row(p.ProductID)))
FROM AdventureWorksEntities.Products AS p
```

See also

- Entity SQL Reference
- DEREF
- KEY

- REF

DEREF (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Dereferences a reference value and produces the result of that dereference.

Syntax

```
SELECT DEREF ( o.expression ) FROM Table AS o;
```

Arguments

`expression` Any valid query expression that returns a collection.

Return Value

The value of the entity that is referenced.

Remarks

The DEREF operator dereferences a reference value and produces the result of that dereference. For example, if `r` is a reference of type ref<T>, `Deref(r)` is an expression of type `T` that yields the entity referenced by `r`. If the reference value is null, or is dangling (that is, the target of the reference does not exist), the result of the DEREF operator is null.

Example

The following Entity SQL query uses the DEREF operator to dereference a reference value and produce the result of that dereference. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns PrimitiveType Results.
2. Pass the following query as an argument to the ExecutePrimitiveTypeQuery method:

```
SELECT VALUE DEREF(REF(p)).Name
FROM AdventureWorksEntities.Products AS p
```

See also

- Entity SQL Reference
- REF
- CREATEREF
- KEY
- Nullable Structured Types

EXCEPT (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Returns a collection of any distinct values from the query expression to the left of the EXCEPT operand that are not also returned from the query expression to the right of the EXCEPT operand. All expressions must be of the same type or of a common base or derived type as `expression`.

Syntax

```
expression EXCEPT expression
```

Arguments

`expression` Any valid query expression that returns a collection to compare with the collection returned from another query expression.

Return Value

A collection of the same type or of a common base or derived type as `expression`.

Remarks

EXCEPT is one of the Entity SQL set operators. All Entity SQL set operators are evaluated from left to right. The following table shows the precedence of the Entity SQL set operators.

PRECEDENCE	OPERATORS
Highest	INTERSECT
	UNION UNION ALL
	EXCEPT
Lowest	EXISTS OVERLAPS FLATTEN SET

Example

The following Entity SQL query uses the EXCEPT operator to return a collection of any distinct values from two query expressions. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.

2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
(SELECT product FROM AdventureWorksEntities.Products AS product
    WHERE product.ListPrice  > @price1 ) except
    (select product FROM AdventureWorksEntities.Products AS product
    WHERE product.ListPrice > @price2)
```

See also

- Entity SQL Reference

EXISTS (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Determines if a collection is empty.

Syntax

```
[NOT] EXISTS ( expression )
```

Arguments

`expression` Any valid expression that returns a collection.

NOT Specifies that the result of EXISTS be negated.

Return Value

`true` if the collection is not empty; otherwise, `false`.

Remarks

EXISTS is one of the Entity SQL set operators. All Entity SQL set operators are evaluated from left to right. For precedence information for the Entity SQL set operators, see EXCEPT.

Example

The following Entity SQL query uses the EXISTS operator to determine whether the collection is empty. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE name FROM AdventureWorksEntities.Products
    AS name WHERE exists(SELECT A FROM AdventureWorksEntities.Products
    AS A WHERE A.ListPrice < @price1)
```

See also

- Entity SQL Reference

FLATTEN (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Converts a collection of collections into a flattened collection. The new collection contains all the same elements as the old collection, but without a nested structure.

Syntax

```
FLATTEN ( collection )
```

Arguments

`collection` Any valid expression that returns a collection of value collections to flatten into a single collection.

Remarks

`FLATTEN` is one of the Entity SQL set operators. All Entity SQL set operators are evaluated from left to right. See EXCEPT for precedence information for the Entity SQL set operators.

Example

The following Entity SQL query uses the `FLATTEN` operator to convert a collection of collections into a flattened collection. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
FLATTEN(SELECT VALUE c.SalesOrderHeaders From
    AdventureWorksEntities.Contacts AS c)
```

See also

- Entity SQL Reference

FROM (Entity SQL)

11/8/2022 • 7 minutes to read • Edit Online

Specifies the collection used in SELECT statements.

Syntax

```
FROM expression [ ,...n ] AS C
```

Arguments

`expression`
Any valid query expression that yields a collection to use as a source in a `SELECT` statement.

Remarks

A `FROM` clause is a comma-separated list of one or more `FROM` clause items. The `FROM` clause can be used to specify one or more sources for a `SELECT` statement. The simplest form of a `FROM` clause is a single query expression that identifies a collection and an alias used as the source in a `SELECT` statement, as illustrated in the following example:

```
FROM C as c
```

FROM Clause Items

Each `FROM` clause item refers to a source collection in the Entity SQL query. Entity SQL supports the following classes of `FROM` clause items: simple `FROM` clause items, `JOIN FROM` clause items, and `APPLY FROM` clause items. Each of these `FROM` clause items is described in more detail in the following sections.

Simple FROM Clause Item

The simplest `FROM` clause item is a single expression that identifies a collection and an alias. The expression can simply be an entity set, or a subquery, or any other expression that is a collection type. The following is an example:

```
LOB.Customers as c
```

The alias specification is optional. An alternate specification of the above from clause item could be the following:

```
LOB.Customers
```

If no alias is specified, Entity SQL attempts to generate an alias based on the collection expression.

JOIN FROM Clause Item

A `JOIN FROM` clause item represents a join between two `FROM` clause items. Entity SQL supports cross joins, inner joins, left and right outer joins, and full outer joins. All these joins are supported similar to the way that they are supported in Transact-SQL. As in Transact-SQL, the two `FROM` clause items involved in the `JOIN` must

be independent. That is, they cannot be correlated. A `CROSS APPLY` or `OUTER APPLY` can be used for these cases.

Cross Joins

A `CROSS JOIN` query expression produces the Cartesian product of the two collections, as illustrated in the following example:

```
FROM C AS c CROSS JOIN D as d
```

Inner Joins

An `INNER JOIN` produces a constrained Cartesian product of the two collections, as illustrated in the following example:

```
FROM C AS c [INNER] JOIN D AS d ON e
```

The previous query expression processes a combination of every element of the collection on the left paired against every element of the collection on the right, where the `ON` condition is true. If no `ON` condition is specified, an `INNER JOIN` degenerates to a `CROSS JOIN`.

Left Outer Joins and Right Outer Joins

An `OUTER JOIN` query expression produces a constrained Cartesian product of the two collections, as illustrated in the following example:

```
FROM C AS c LEFT OUTER JOIN D AS d ON e
```

The previous query expression processes a combination of every element of the collection on the left paired against every element of the collection on the right, where the `ON` condition is true. If the `ON` condition is false, the expression still processes a single instance of the element on the left paired against the element on the right, with the value null.

A `RIGHT OUTER JOIN` may be expressed in a similar manner.

Full Outer Joins

An explicit `FULL OUTER JOIN` produces a constrained Cartesian product of the two collections as illustrated in the following example:

```
FROM C AS c FULL OUTER JOIN D AS d ON e
```

The previous query expression processes a combination of every element of the collection on the left paired against every element of the collection on the right, where the `ON` condition is true. If the `ON` condition is false, the expression still processes one instance of the element on the left paired against the element on the right, with the value null. It also processes one instance of the element on the right paired against the element on the left, with the value null.

> **NOTE**
> To preserve compatibility with SQL-92, in Transact-SQL the OUTER keyword is optional. Therefore, `LEFT JOIN`, `RIGHT JOIN`, and `FULL JOIN` are synonyms for `LEFT OUTER JOIN`, `RIGHT OUTER JOIN`, and `FULL OUTER JOIN`.

APPLY Clause Item

Entity SQL supports two kinds of `APPLY`: `CROSS APPLY` and `OUTER APPLY`.

A `CROSS APPLY` produces a unique pairing of each element of the collection on the left with an element of the collection produced by evaluating the expression on the right. With a `CROSS APPLY`, the expression on the right is functionally dependent on the element on the left, as illustrated in the following associated collection example:

```
SELECT c, f FROM C AS c CROSS APPLY c.Assoc AS f
```

The behavior of `CROSS APPLY` is similar to the join list. If the expression on the right evaluates to an empty

collection, the `CROSS APPLY` produces no pairings for that instance of the element on the left.

An `OUTER APPLY` resembles a `CROSS APPLY`, except a pairing is still produced even when the expression on the right evaluates to an empty collection. The following is an example of an `OUTER APPLY`:

```
SELECT c, f FROM C AS c OUTER APPLY c.Assoc AS f
```

> **NOTE**
> Unlike in Transact-SQL, there is no need for an explicit unnest step in Entity SQL.

> **NOTE**
> `CROSS` and `OUTER APPLY` operators were introduced in SQL Server 2005. In some cases, the query pipeline might produce Transact-SQL that contains `CROSS APPLY` and/or `OUTER APPLY` operators. Because some backend providers, including versions of SQL Server earlier than SQL Server 2005, do not support these operators, such queries cannot be executed on these backend providers.
>
> Some typical scenarios that might lead to the presence of `CROSS APPLY` and/or `OUTER APPLY` operators in the output query are the following: a correlated subquery with paging; AnyElement over a correlated subquery or over a collection produced by navigation; LINQ queries that use grouping methods that accept an element selector; a query in which a `CROSS APPLY` or an `OUTER APPLY` are explicitly specified; a query that has a `DEREF` construct over a `REF` construct.

Multiple Collections in the FROM Clause

The `FROM` clause can contain more than one collection separated by commas. In these cases, the collections are assumed to be joined together. Think of these as an n-way CROSS JOIN.

In the following example, `C` and `D` are independent collections, but `c.Names` is dependent on `C`.

```
FROM C AS c, D AS d, c.Names AS e
```

The previous example is logically equivalent to the following example:

```
FROM (C AS c JOIN D AS d) CROSS APPLY c.Names AS e
```

Left Correlation

Items in the `FROM` clause can refer to items specified in earlier clauses. In the following example, `C` and `D` are independent collections, but `c.Names` is dependent on `C`:

```
from C as c, D as d, c.Names as e
```

This is logically equivalent to:

```
from (C as c join D as d) cross apply c.Names as e
```

Semantics

Logically, the collections in the `FROM` clause are assumed to be part of an `n`-way cross join (except in the case of a 1-way cross join). Aliases in the `FROM` clause are processed left to right, and are added to the current scope for later reference. The `FROM` clause is assumed to produce a multiset of rows. There will be one field for each

item in the `FROM` clause that represents a single element from that collection item.

The `FROM` clause logically produces a multiset of rows of type Row(c, d, e) where fields c, d, and e are assumed to be of the element type of `C`, `D`, and `c.Names`.

Entity SQL introduces an alias for each simple `FROM` clause item in scope. For example, in the following FROM clause snippet, The names introduced into scope are c, d, and e.

```
from (C as c join D as d) cross apply c.Names as e
```

In Entity SQL (unlike Transact-SQL), the `FROM` clause only introduces the aliases into scope. Any references to columns (properties) of these collections must be qualified with the alias.

Pulling Up Keys from Nested Queries

Certain types of queries that require pulling up keys from a nested query are not supported. For example, the following query is valid:

```
select c.Orders from Customers as c
```

However, the following query is not valid, because the nested query does not have any keys:

```
select {1} from {2, 3}
```

See also

- Entity SQL Reference
- Query Expressions
- Nullable Structured Types

FUNCTION (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Defines a function in the scope of an Entity SQL query command.

Syntax

```
FUNCTION function-name
( [ { parameter_name <type_definition>
        [ ,...n ]
  ]
) AS ( function_expression )

<type_definition>::=
    { data_type | COLLECTION ( <type_definition> )
                | REF ( data_type )
                | ROW ( row_expression )
        }
```

Arguments

`function-name`
Name of the function.

`parameter-name`
Name of a parameter in the function.

`function_expression`
A valid Entity SQL expression that is the function. The command in the function can act on `parameter_name` parameters passed to the function.

`data_type`
Name of a supported type.

COLLECTION (<type_definition`>`)
An expression that returns a collection of supported types, rows, or references.

REF (`data_type`)
An expression that returns a reference to an entity type.

ROW (`row_expression`)
An expression that returns anonymous, structurally typed records from one or more values. For more information, see ROW.

Remarks

Multiple functions with the same name can be declared inline, as long as the function signatures are different. For more information, see Function Overload Resolution.

An inline function can be called in an Entity SQL command only after it has been defined in that command. However, an inline function can be called inside another inline function either before or after the called function has been defined. In the following example, function A calls function B before function B is defined:

```
Function A() as ('A calls B. ' + B())
```

```
Function B() as ('B was called.')
```

```
A()
```

For more information, see How to: Call a User-Defined Function.

Functions can also be declared in the model itself. Functions declared in the model are executed in the same way as functions declared inline in the command. For more information, see User-Defined Functions.

Example 1

The following Entity SQL command defines a function `Products` that takes an integer value to filter the returned products.

```
USING Microsoft.Samples.Entity;
FUNCTION Products(listPrice Int32) AS
(
SELECT VALUE p FROM AdventureWorksEntities.Products AS p
    WHERE p.ListPrice >= listPrice
)
select p FROM Products(@price) AS p
```

Example 2

The following Entity SQL command defines a function `StringReturnsCollection` that takes a collection of strings to filter the returned contacts.

```
USING Microsoft.Samples.Entity;
FUNCTION GetSpecificContacts(Ids Collection(Int32)) AS
(
SELECT VALUE id FROM Ids AS id WHERE id < @price
)
GetSpecificContacts(SELECT VALUE c.ContactID
    FROM AdventureWorksEntities.Contacts AS c)
```

See also

- Entity SQL Reference
- Entity SQL Language

GROUP BY (Entity SQL)

11/8/2022 • 3 minutes to read • Edit Online

Specifies groups into which objects returned by a query (SELECT) expression are to be placed.

Syntax

```
[ GROUP BY aliasedExpression [ ,...n ] ]
```

Arguments

`aliasedExpression` Any valid query expression on which grouping is performed. `expression` can be a property or a non-aggregate expression that references a property returned by the FROM clause. Each expression in a GROUP BY clause must evaluate to a type that can be compared for equality. These types are generally scalar primitives such as numbers, strings, and dates. You cannot group by a collection.

Remarks

If aggregate functions are included in the SELECT clause <select list>, GROUP BY calculates a summary value for each group. When GROUP BY is specified, either each property name in any nonaggregate expression in the select list should be included in the GROUP BY list, or the GROUP BY expression must exactly match the select list expression.

> **NOTE**
> If the ORDER BY clause is not specified, groups returned by the GROUP BY clause are not in any particular order. To specify a particular ordering of the data, we recommend that you always use the ORDER BY clause.

When a GROUP BY clause is specified, either explicitly or implicitly (for example, by a HAVING clause in the query), the current scope is hidden, and a new scope is introduced.

The SELECT clause, the HAVING clause, and the ORDER BY clause will no longer be able to refer to element names specified in the FROM clause. You can only refer to the grouping expressions themselves. To do this, you can assign new names (aliases) to each grouping expression. If no alias is specified for a grouping expression, Entity SQL tries to generate one by using the alias generation rules, as illustrated in the following example.

```
SELECT g1, g2, ...gn FROM c as c1
```

```
GROUP BY e1 as g1, e2 as g2, ...en as gn
```

Expressions in the GROUP BY clause cannot refer to names defined earlier in the same GROUP BY clause.

In addition to grouping names, you can also specify aggregates in the SELECT clause, HAVING clause, and the ORDER BY clause. An aggregate contains an expression that is evaluated for each element of the group. The aggregate operator reduces the values of all these expressions (usually, but not always, into a single value). The aggregate expression can make references to the original element names visible in the parent scope, or to any of the new names introduced by the GROUP BY clause itself. Although the aggregates appear in the SELECT clause, HAVING clause, and ORDER BY clause, they are actually evaluated under the same scope as the grouping expressions, as illustrated in the following example.

`SELECT name, sum(o.Price * o.Quantity) as total`

`FROM orderLines as o`

`GROUP BY o.Product as name`

This query uses the GROUP BY clause to produce a report of the cost of all products ordered, broken down by product. It gives the name `name` to the product as part of the grouping expression, and then references that name in the SELECT list. It also specifies the aggregate `sum` in the SELECT list that internally references the price and quantity of the order line.

Each GROUP By key expression must have at least one reference to the input scope:

```
SELECT FROM Persons as P
GROUP BY Q + P   -- GOOD
GROUP BY Q   -- BAD
GROUP BY 1   -- BAD, a constant is not allowed
```

For an example of using GROUP BY, see HAVING.

Example

The following Entity SQL query uses the GROUP BY operator to specify groups into which objects are returned by a query. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns PrimitiveType Results.
2. Pass the following query as an argument to the `ExecutePrimitiveTypeQuery` method:

```
SELECT VALUE name FROM AdventureWorksEntities.Products
    AS P GROUP BY P.Name HAVING MAX(P.ListPrice) > @price
```

See also

- Entity SQL Reference
- Query Expressions

GROUPPARTITION (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Returns a collection of argument values that are projected off the current group partition to which the aggregate is related. The `GroupPartition` aggregate is a group-based aggregate and has no collection-based form.

Syntax

```
GROUPPARTITION( [ALL|DISTINCT] expression )
```

Arguments

`expression` Any Entity SQL expression.

Remarks

The following query produces a list of products and a collection of order line quantities per each product:

```
SELECT p, GroupPartition(ol.Quantity) FROM LOB.OrderLines AS ol
  GROUP BY ol.Product AS p
```

The following two queries are semantically equal:

```
SELECT p, Sum(GroupPartition(ol.Quantity)) FROM LOB.OrderLines AS ol
  GROUP BY ol.Product AS p
SELET p, Sum(ol.Quantity) FROM LOB.OrderLines AS ol
  group by ol.Product as p
```

The `GROUPPARTITION` operator can be used in conjunction with user-defined aggregate functions.

`GROUPPARTITION` is a special aggregate operator that holds a reference to the grouped input set. This reference can be used anywhere in the query where GROUP BY is in scope. For example:

```
SELECT p, GroupPartition(ol.Quantity) FROM LOB.OrderLines AS ol GROUP BY ol.Product AS p
```

With a regular `GROUP BY`, the results of the grouping are hidden. You can only use the results in an aggregate function. In order to see the results of the grouping, you have to correlate the results of the grouping and the input set by using a subquery. The following two queries are equivalent:

```
SELET p, (SELECT q FROM GroupPartition(ol.Quantity) AS q) FROM LOB.OrderLines AS ol GROUP BY ol.Product AS p
SELECT p, (SELECT ol.Quantity AS q FROM LOB.OrderLines AS ol2 WHERE ol2.Product = p) FROM LOB.OrderLines AS
ol GROUP BY ol.Product AS p
```

As seen from the example, the GROUPPARTITION aggregate operator makes it easier to get a reference to the input set after the grouping.

The GROUPPARTITION operator can specify any Entity SQL expression in the operator input when you use the `expression` parameter.

For instance all of the following input expressions to the group partition are valid:

```
SELECT groupkey, GroupPartition(b) FROM {1,2,3} AS a INNER JOIN {4,5,6} AS b ON true GROUP BY a AS groupkey
SELECT groupkey, GroupPartition(1) FROM {1,2,3} AS a INNER JOIN {4,5,6} AS b ON true GROUP BY a AS groupkey
SELECT groupkey, GroupPartition(a + b) FROM {1,2,3} AS a INNER JOIN {4,5,6} AS b ON true GROUP BY a AS
groupkey
SELECT groupkey, GroupPartition({a + b}) FROM {1,2,3} AS a INNER JOIN {4,5,6} AS b ON true GROUP BY a AS
groupkey
SELECT groupkey, GroupPartition({42}) FROM {1,2,3} AS a INNER JOIN {4,5,6} AS b ON true GROUP BY a AS
groupkey
SELECT groupkey, GroupPartition(b > a) FROM {1,2,3} AS a INNER JOIN {4,5,6} AS b ON true GROUP BY a AS
groupkey
```

Example

The following example shows how to use the GROUPPARTITION clause with the GROUP BY clause. The GROUP BY clause groups `SalesOrderHeader` entities by their `Contact`. The GROUPPARTITION clause then projects the `TotalDue` property for each group, resulting in a collection of decimals.

```
USING Microsoft.Samples.Entity
Function MyAvg(dues Collection(Decimal)) AS
(
    Avg(SELECT value due FROM dues AS due WHERE due > @price)
)
SELECT TOP(10) contactID, MyAvg(GroupPartition(order.TotalDue))
FROM AdventureWorksEntities.SalesOrderHeaders AS order
GROUP BY order.Contact.ContactID AS contactID;
```

HAVING (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Specifies a search condition for a group or an aggregate.

Syntax

```
[ HAVING search_condition ]
```

Arguments

`search_condition`

Specifies the search condition for the group or the aggregate to meet. When HAVING is used with GROUP BY ALL, the HAVING clause overrides ALL.

Remarks

The HAVING clause is used to specify an additional filtering condition on the result of a grouping. If no GROUP BY clause is specified in the query expression, an implicit single-set group is assumed.

> **NOTE**
> HAVING can be used only with the SELECT statement. When GROUP BY is not used, HAVING behaves like a WHERE clause.

The HAVING clause works like the WHERE clause except that it is applied after the GROUP BY operation. This means that the HAVING clause can only make references to grouping aliases and aggregates, as illustrated in the following example:

```
SELECT Name, SUM(o.Price * o.Quantity) AS Total FROM orderLines AS o GROUP BY o.Product AS Name
HAVING SUM(o.Quantity) > 1
```

The previous restricts the groups to only those that include more than one product.

Example

The following Entity SQL query uses the HAVING and GROUP BY operators to specify a search condition for a group or an aggregate. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns PrimitiveType Results.
2. Pass the following query as an argument to the `ExecutePrimitiveTypeQuery` method:

```
SELECT VALUE name FROM AdventureWorksEntities.Products
    AS P GROUP BY P.Name HAVING MAX(P.ListPrice) > @price
```

See also

- Entity SQL Reference
- Query Expressions

KEY (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Extracts the key of a reference or of an entity expression.

Syntax

```
KEY(createref_expression)
```

Remarks

An entity key contains the key values in the correct order of the specified entity or entity reference. Because multiple entity sets can be based on the same type, the same key might appear in each entity set. To get a unique reference, use `REF`. The return type of the KEY operator is a row type that includes one field for each key of the entity, in the same order.

In the following example, the key operator is passed a reference to the BadOrder entity, and returns the key portion of that reference. In this case, a record type with exactly one field corresponding to the `Id` property.

```
select Key( CreateRef(LOB.BadOrders, row(o.Id)) )
from LOB.Orders as o
```

Example

The following Entity SQL query uses the KEY operator to extract the key portion of an expression with type reference. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE Key(CreateRef(AdventureWorksEntities.Products,
    row(p.ProductID))) FROM AdventureWorksEntities.Products AS p
```

See also

- Entity SQL Reference
- CREATEREF
- REF
- DEREF

IN (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Determines whether a value matches any value in a collection.

Syntax

```
value [ NOT ] IN expression
```

Arguments

`value`
Any valid expression that returns the value to match.

[NOT]
Specifies that the `Boolean` result of IN be negated.

`expression`
Any valid expression that returns the collection to test for a match. All expressions must be of the same type or of a common base or derived type as `value`.

Return Value

`true` if the value is found in the collection; null if the value is null or the collection is null; otherwise, `false`. Using NOT IN negates the results of IN.

Example

The following Entity SQL query uses the IN operator to determine whether a value matches any value in a collection. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE product FROM AdventureWorksEntities.Products
    AS product WHERE product.ListPrice IN {125, 300}
```

See also

- Entity SQL Reference

INTERSECT (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Returns a collection of any distinct values that are returned by both the query expressions on the left and right sides of the INTERSECT operand. All expressions must be of the same type or of a common base or derived type as `expression`.

Syntax

```
expression INTERSECT expression
```

Arguments

`expression` Any valid query expression that returns a collection to compare with the collection returned from another query expression.

Return Value

A collection of the same type or of a common base or derived type as `expression`.

Remarks

INTERSECT is one of the Entity SQL set operators. All Entity SQL set operators are evaluated from left to right. For precedence information for the Entity SQL set operators, see EXCEPT.

Example

The following Entity SQL query uses the INTERSECT operator to return a collection of any distinct values that are returned by both the query expressions on the left and right sides of the INTERSECT operand. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
(SELECT product
    FROM AdventureWorksEntities.Products AS product
    WHERE product.ListPrice > @price1 )
    intersect (SELECT product FROM AdventureWorksEntities.Products AS
    product WHERE product.ListPrice > @price2)
```

See also

- Entity SQL Reference

ISNULL (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Determines if a query expression is null.

Syntax

```
expression IS [ NOT ] NULL
```

Arguments

`expression` Any valid query expression. Cannot be a collection, have collection members, or a record type with collection type properties.

NOT Negates the EDM.Boolean result of IS NULL.

Return Value

`true` if `expression` returns null; otherwise, `false`.

Remarks

Use `IS NULL` to determine if the element of an outer join is null:

```
select c
      from LOB.Customers as c left outer join LOB.Orders as o
                              on c.ID = o.CustomerID
      where o is not null and o.OrderQuantity = @x
```

Use `IS NULL` to determine if a member has an actual value:

```
select c from LOB.Customer as c where c.DOB is not null
```

The following table shows the behavior of `IS NULL` over some patterns. All exceptions are thrown from the client side before the provider gets invoked:

PATTERN	BEHAVIOR
null IS NULL	Returns `true`.
TREAT (null AS EntityType) IS NULL	Returns `true`.
TREAT (null AS ComplexType) IS NULL	Throws an error.
TREAT (null AS RowType) IS NULL	Throws an error.
EntityType IS NULL	Returns `true` or `false`.

PATTERN	BEHAVIOR
ComplexType IS NULL	Throws an error.
RowType IS NULL	Throws an error.

Example

The following Entity SQL query uses the IS NOT NULL operator to determine if a query expression is not null. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE product FROM AdventureWorksEntities.Products
    AS product WHERE product.Color IS NOT NULL
```

See also

- Entity SQL Reference

ISOF (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Determines whether the type of an expression is of the specified type or one of its subtypes.

Syntax

```
expression IS [ NOT ] OF ( [ ONLY ] type )
```

Arguments

`expression` Any valid query expression to determine the type of.

NOT Negates the EDM.Boolean result of IS OF.

ONLY Specifies that IS OF returns `true` only if `expression` is of type `type` and not any of one its subtypes.

`type` The type to test `expression` against. The type must be namespace-qualified.

Return Value

`true` if `expression` is of type T and T is either a base type, or a derived type of `type`; null if `expression` is null at run time; otherwise, `false`.

Remarks

The expressions `expression IS NOT OF (type)` and `expression IS NOT OF (ONLY type)` are syntactically equivalent to `NOT (expression IS OF (type))` and `NOT (expression IS OF (ONLY type))`, respectively.

The following table shows the behavior of `IS OF` operator over some typical- and corner patterns. All exceptions are thrown from the client side before the provider gets invoked:

PATTERN	BEHAVIOR
null IS OF (EntityType)	Throws
null IS OF (ComplexType)	Throws
null IS OF (RowType)	Throws
TREAT (null AS EntityType) IS OF (EntityType)	Returns DBNull
TREAT (null AS ComplexType) IS OF (ComplexType)	Throws
TREAT (null AS RowType) IS OF (RowType)	Throws
EntityType IS OF (EntityType)	Returns true/false
ComplexType IS OF (ComplexType)	Throws

PATTERN	BEHAVIOR
RowType IS OF (RowType)	Throws

Example

The following Entity SQL query uses the IS OF operator to determine the type of a query expression, and then uses the TREAT operator to convert an object of the type Course to a collection of objects of the type OnsiteCourse. The query is based on the School Model.

[!code-sql[DP EntityServices Concepts#TREAT_ISOF]~/samples/snippets/tsql/VS_Snippets_Data/dp entityservices concepts/tsql/entitysql.sql#treat_isof)]

See also

- Entity SQL Reference

LIKE (Entity SQL)

11/8/2022 • 3 minutes to read • Edit Online

Determines whether a specific character `String` matches a specified pattern.

Syntax

```
match [NOT] LIKE pattern [ESCAPE escape]
```

Arguments

`match` An Entity SQL expression that evaluates to a `String`.

`pattern` A pattern to match to the specified `String`.

`escape` An escape character.

NOT Specifies that the result of LIKE be negated.

Return Value

`true` if the `string` matches the pattern; otherwise, `false`.

Remarks

Entity SQL expressions that use the LIKE operator are evaluated in much the same way as expressions that use equality as the filter criteria. However, Entity SQL expressions that use the LIKE operator can include both literals and wildcard characters.

The following table describes the syntax of the pattern `string`.

WILDCARD CHARACTER	DESCRIPTION	EXAMPLE
%	Any `string` of zero or more characters.	`title like '%computer%'` finds all titles with the word `"computer"` anywhere in the title.
_ (underscore)	Any single character.	`firstname like '_ean'` finds all four-letter first names that end with `"ean"`," such as Dean or Sean.
[ ]	Any single character in the specified range ([a-f]) or set ([abcdef]).	`lastname like '[C-P]arsen'` finds last names ending with "arsen" and beginning with any single character between C and P, such as Carsen or Larsen.
[^]	Any single character not in the specified range ([^a-f]) or set ([^abcdef]).	`lastname like 'de[^l]%'` finds all last names that begin with "de" and do not include "l" as the following letter.

> **NOTE**
> The Entity SQL LIKE operator and ESCAPE clause cannot be applied to `System.DateTime` or `System.Guid` values.

LIKE supports ASCII pattern matching and Unicode pattern matching. When all parameters are ASCII characters, ASCII pattern matching is performed. If one or more of the arguments are Unicode, all arguments are converted to Unicode and Unicode pattern matching is performed. When you use Unicode with LIKE, trailing blanks are significant; however, for non-Unicode, trailing blanks are not significant. The pattern string syntax of Entity SQL is the same as that of Transact-SQL.

A pattern can include regular characters and wildcard characters. During pattern matching, regular characters must exactly match the characters specified in the character `string`. However, wildcard characters can be matched with arbitrary fragments of the character string. When it is used with wildcard characters, the LIKE operator is more flexible than the = and != string comparison operators.

> **NOTE**
> You may use provider-specific extensions if you target a specific provider. However, such constructs may be treated differently by other providers, for example. SqlServer supports [first-last] and [^first-last] patterns where the former matches exactly one character between first and last, and the latter matches exactly one character that is not between first and last.

Escape

By using the ESCAPE clause, you can search for character strings that include one or more of the special wildcard characters described in the table in the previous section. For example, assume several documents include the literal "100%" in the title and you want to search for all of those documents. Because the percent (%) character is a wildcard character, you must escape it using the Entity SQL ESCAPE clause to successfully execute the search. The following is an example of this filter.

```
"title like '%100!%%' escape '!'"
```

In this search expression, the percent wildcard character (%) immediately following the exclamation point character (!) is treated as a literal, instead of as a wildcard character. You can use any character as an escape character except for the Entity SQL wildcard characters and the square bracket (`[ ]`) characters. In the previous example, the exclamation point (!) character is the escape character.

Example

The following two Entity SQL queries use the LIKE and ESCAPE operators to determine whether a specific character string matches a specified pattern. The first query searches for the `Name` that starts with the characters `Down_`. This query uses the ESCAPE option because the underscore (`_`) is a wildcard character. Without specifying the ESCAPE option, the query would search for any `Name` values that start with the word `Down` followed by any single character other than the underscore character. The queries are based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns PrimitiveType Results.
2. Pass the following query as an argument to the `ExecutePrimitiveTypeQuery` method:

```
-- LIKE and ESCAPE
-- If an AdventureWorksEntities.Products contained a Name
-- with the value 'Down_Tube', the following query would find that
-- value.
Select value P.Name FROM AdventureWorksEntities.Products AS P
WHERE P.Name LIKE 'DownA_%' ESCAPE 'A'

-- LIKE
Select value P.Name FROM AdventureWorksEntities.Products AS P
WHERE P.Name LIKE 'BB%'
```

See also

- Entity SQL Reference

LIMIT (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Physical paging can be performed by using LIMIT sub-clause in ORDER BY clause. LIMIT can not be used separately from ORDER BY clause.

Syntax

```
[ LIMIT n ]
```

Arguments

`n`

The number of items that will be selected.

If a LIMIT expression sub-clause is present in an ORDER BY clause, the query will be sorted according to the sort specification and the resulting number of rows will be restricted by the LIMIT expression. For instance, LIMIT 5 will restrict the result set to 5 instances or rows. LIMIT is functionally equivalent to TOP with the exception that LIMIT requires ORDER BY clause to be present. SKIP and LIMIT can be used independently along with ORDER BY clause.

> **NOTE**
> An Entity Sql query will be considered invalid if TOP modifier and SKIP sub-clause is present in the same query expression. The query should be rewritten by changing TOP expression to LIMIT expression.

Example

The following Entity SQL query uses the ORDER BY operator with LIMIT to specify the sort order used on objects returned in a SELECT statement. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE p FROM AdventureWorksEntities.Products AS p
ORDER BY p.ListPrice LIMIT(@limit)
```

See also

- ORDER BY
- How to: Page Through Query Results
- Paging
- TOP

MULTISET (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Creates an instance of a multiset from a list of values. All the values in the MULTISET constructor must be of a compatible type `T`. Empty multiset constructors are not allowed.

Syntax

```
MULTISET ( expression [{, expression }] )
-- or
{ expression [{, expression }] }
```

Arguments

`expression` Any valid list of values.

Return Value

A collection of type MULTISET<T>.

Remarks

Entity SQL provides three kinds of constructors: row constructors, object constructors, and multiset (or collection) constructors. For more information, see Constructing Types.

The multiset constructor creates an instance of a multiset from a list of values. All the values in the constructor must be of a compatible type.

For example, the following expression creates a multiset of integers.

`MULTISET(1, 2, 3)`

`{1, 2, 3}`

> **NOTE**
>
> Nested multiset literals are only supported when a wrapping multiset has a single multiset element; for example, `{{1, 2, 3}}`. When the wrapping multiset has multiple multiset elements (for example, `{{1, 2}, {3, 4}}`), nested multiset literals are not supported.

Example

The following Entity SQL query uses the MULTISET operator to create an instance of a multiset from a list of values. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE product FROM AdventureWorksEntities.Products
    AS product
WHERE product.ListPrice IN MultiSet (@price1, @price2)
```

See also

- Constructing Types
- Entity SQL Reference

Named Type Constructor (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Used to create instances of conceptual model nominal types such as Entity or Complex types.

Syntax

```
[{identifier. }] identifier( [expression [{, expression }]] )
```

Arguments

`identifier`
Value that is a simple or quoted identifier. For more information see, Identifiers

`expression`
Attributes of the type that are assumed to be in the same order as they appear in the declaration of the type.

Return Value

Instances of named complex types and entity types.

Remarks

The following examples show how to construct nominal and complex types:

The expression below creates an instance of a `Person` type:

```
Person("abc", 12)
```

The expression below creates an instance of a complex type:

```
MyModel.ZipCode('98118', '4567')
```

The expression below creates an instance of a nested complex type:

```
MyModel.AddressInfo('My street address', 'Seattle', 'WA', MyModel.ZipCode('98118', '4567'))
```

The expression below creates an instance of an entity with a nested complex type:

```
MyModel.Person("Bill", MyModel.AddressInfo('My street address', 'Seattle', 'WA', MyModel.ZipCode('98118',
'4567')))
```

The following example shows how to initialize a property of a complex type to null:

```
MyModel.ZipCode('98118', null)
```

Example

The following Entity SQL query uses the named type constructor to create an instance of a conceptual model type. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE AdventureWorksModel.SalesOrderDetail
    (o.SalesOrderID, o.SalesOrderDetailID, o.CarrierTrackingNumber,
    o.OrderQty, o.ProductID, o.SpecialOfferID, o.UnitPrice,
    o.UnitPriceDiscount, o.LineTotal, o.rowguid, o.ModifiedDate)
FROM AdventureWorksEntities.SalesOrderDetails AS o
```

See also

- Constructing Types
- Entity SQL Reference

NAVIGATE (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Navigates over the relationship established between entities.

Syntax

```
navigate(instance-expression, [relationship-type], [to-end [, from-end] ])
```

Arguments

`instance-expression` An instance of an entity.

`relationship-type` The type name of the relationship, from the conceptual schema definition language (CSDL) file. The `relationship-type` is qualified as <namespace>.<relationship type name>.

`to` The end of the relationship.

`from` The beginning of the relationship.

Return Value

If the cardinality of the to end is 1, the return value will be `Ref<T>`. If the cardinality of the to end is n, the return value will be `Collection<Ref<T>>`.

Remarks

Relationships are first-class constructs in the Entity Data Model (EDM). Relationships can be established between two or more entity types, and users can navigate over the relationship from one end (entity) to another. `from` and `to` are conditionally optional when there is no ambiguity in name resolution within the relationship.

NAVIGATE is valid in O and C space.

The general form of a navigation construct is the following:

navigate(`instance-expression` , `relationship-type` , [`to-end` [, `from-end`]])

For example:

```
Select o.Id, navigate(o, OrderCustomer, Customer, Order)
From LOB.Orders as o
```

Where OrderCustomer is the `relationship`, and Customer and Order are the `to-end` (customer) and `from-end` (order) of the relationship. If OrderCustomer was a n:1 relationship, then the result type of the navigate expression is Ref<Customer>.

The simpler form of this expression is the following:

```
Select o.Id, navigate(o, OrderCustomer)
From LOB.Orders as o
```

Similarly, in a query of the following form, The navigate expression would produce a Collection<Ref<Order>>.

```
Select c.Id, navigate(c, OrderCustomer, Order, Customer)
From LOB.Customers as c
```

The instance-expression must be an entity/ref type.

Example

The following Entity SQL query uses the NAVIGATE operator to navigate over the relationship established between Address and SalesOrderHeader entity types. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT address.AddressID, (SELECT VALUE DEREF(soh)
FROM NAVIGATE(address,
    AdventureWorksModel.FK_SalesOrderHeader_Address_BillToAddressID)
    AS soh)
FROM AdventureWorksEntities.Addresses AS address
```

See also

- Entity SQL Reference
- How to: Navigate Relationships with Navigate Operator

OFTYPE (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Returns a collection of objects from a query expression that is of a specific type.

Syntax

```
OFTYPE ( expression, [ONLY] test_type )
```

Arguments

`expression` Any valid query expression that returns a collection of objects.

`test_type` The type to test each object returned by `expression` against. The type must be qualified by a namespace.

Return Value

A collection of objects that are of type `test_type`, or a base type or derived type of `test_type`. If ONLY is specified, only instances of the `test_type` or an empty collection will be returned.

Remarks

An `OFTYPE` expression specifies a type expression that is issued to perform a type test against each element of a collection. The `OFTYPE` expression produces a new collection of the specified type containing only those elements that were either equivalent to that type or a sub-type of it.

An `OFTYPE` expression is an abbreviation of the following query expression:

```
select value treat(t as T) from ts as t where t is of (T)
```

Given that a Manager is a subtype of Employee, the following expression produces a collection of only managers from a collection of employees:

```
OfType(employees, NamespaceName.Manager)
```

It is also possible to up cast a collection using the type filter:

```
OfType(executives, NamespaceName.Manager)
```

Since all executives are managers, the resulting collection still contains all the original executives, though the collection is now typed as a collection of managers.

The following table shows the behavior of the `OFTYPE` operator over some patterns. All exceptions are thrown from the client side before the provider is invoked:

PATTERN	BEHAVIOR
OFTYPE(Collection(EntityType), EntityType)	Collection(EntityType)
OFTYPE(Collection(ComplexType), ComplexType)	Throws
OFTYPE(Collection(RowType), RowType)	Throws

Example

The following Entity SQL query uses the OFTYPE operator to return a collection of OnsiteCourse objects from a collection of Course objects. The query is based on the School Model.

```
SELECT onsiteCourse.Location FROM
    OFTYPE(SchoolEntities.Courses, SchoolModel.OnsiteCourse)
    AS onsiteCourse
```

See also

- Entity SQL Reference

ORDER BY (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Specifies the sort order used on objects returned in a SELECT statement.

Syntax

```
[ ORDER BY
   {
      order_by_expression [SKIP n] [LIMIT n]
      [ COLLATE collation_name ]
      [ ASC | DESC ]
   }
   [ ,…n ]
]
```

Arguments

`order_by_expression` Any valid query expression specifying a property on which to sort. Multiple sort expressions can be specified. The sequence of the sort expressions in the ORDER BY clause defines the organization of the sorted result set.

COLLATE {collation_name} Specifies that the ORDER BY operation should be performed according to the collation specified in `collation_name`. COLLATE is applicable only for string expressions.

ASC Specifies that the values in the specified property should be sorted in ascending order, from lowest value to highest value. This is the default.

DESC Specifies that the values in the specified property should be sorted in descending order, from highest value to lowest value.

LIMIT `n` Only the first `n` items will be selected.

SKIP `n` Skips the first `n` items.

Remarks

The ORDER BY clause is logically applied to the result of the SELECT clause. The ORDER BY clause can reference items in the select list by using their aliases. The ORDER BY clause can also reference other variables that are currently in-scope. However, if the SELECT clause has been specified with a DISTINCT modifier, the ORDER BY clause can only reference aliases from the SELECT clause.

```
SELECT c AS c1 FROM cs AS c ORDER BY c1.e1, c.e2
```

Each expression in the ORDER BY clause must evaluate to some type that can be compared for ordered inequality (less than or greater than, and so on). These types are generally scalar primitives such as numbers, strings, and dates. RowTypes of comparable types are also order comparable.

If your code iterates over an ordered set, other than for a top-level projection, the output is not guaranteed to have its order preserved.

In the following sample, order is guaranteed to be preserved:

```
SELECT C1.FirstName, C1.LastName
        FROM AdventureWorks.Contact as C1
        ORDER BY C1.LastName
```

In the following query, ordering of the nested query is ignored:

```
SELECT C2.FirstName, C2.LastName
    FROM (SELECT C1.FirstName, C1.LastName
        FROM AdventureWorks.Contact as C1
        ORDER BY C1.LastName) as C2
```

To have an ordered UNION, UNION ALL, EXCEPT, or INTERSECT operation, use the following pattern:

```
SELECT ...
FROM ( UNION/EXCEPT/INTERSECT operation )
ORDER BY ...
```

Restricted keywords

The following keywords must be enclosed in quotation marks when used in an `ORDER BY` clause:

- CROSS
- FULL
- KEY
- LEFT
- ORDER
- OUTER
- RIGHT
- ROW
- VALUE

Ordering Nested Queries

In the Entity Framework, a nested expression can be placed anywhere in the query; the order of a nested query is not preserved.

The following query will order the results by the last name:

```
SELECT C1.FirstName, C1.LastName
        FROM AdventureWorks.Contact as C1
        ORDER BY C1.LastName
```

In the following query, ordering of the nested query is ignored:

```
SELECT C2.FirstName, C2.LastName
    FROM (SELECT C1.FirstName, C1.LastName
        FROM AdventureWorks.Contact as C1
        ORDER BY C1.LastName) as C2
```

Example

The following Entity SQL query uses the ORDER BY operator to specify the sort order used on objects returned in a SELECT statement. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE p FROM AdventureWorksEntities.Products
    AS p ORDER BY p.ListPrice
```

See also

- Query Expressions
- Entity SQL Reference
- SKIP
- LIMIT
- TOP

OVERLAPS (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Determines whether two collections have common elements.

Syntax

```
expression OVERLAPS expression
```

Arguments

`expression` Any valid query expression that returns a collection to compare with the collection returned from another query expression. All expressions must be of the same type or of a common base or derived type as `expression`.

Return Value

`true` if the two collections have common elements; otherwise, `false`.

Remarks

OVERLAPS provides functionally equivalent to the following:

```
EXISTS ( expression INTERSECT expression )
```

OVERLAPS is one of the Entity SQL set operators. All Entity SQL set operators are evaluated from left to right. For precedence information for the Entity SQL set operators, see EXCEPT.

Example

The following Entity SQL query uses the OVERLAPS operator to determines whether two collections have a common value. The query is based on the AdventureWorks Sales Model. To compile and run this, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT value P FROM AdventureWorksEntities.Products
    AS P WHERE ((SELECT P FROM AdventureWorksEntities.Products
    AS P WHERE P.ListPrice > @price1) overlaps (SELECT P FROM
    AdventureWorksEntities.Products AS P WHERE P.ListPrice < @price2))
```

See also

- Entity SQL Reference

REF (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Returns a reference to an entity instance.

Syntax

```
REF( expression )
```

Arguments

`expression`
Any valid expression that yields an instance of an entity type.

Return Value

A reference to the specified entity instance.

Remarks

An entity reference consists of the entity key and an entity set name. Because different entity sets can be based on the same entity type, a particular entity key can appear in multiple entity sets. However, an entity reference is always unique. If the input expression represents a persisted entity, a reference to this entity will be returned. If the input expression is not a persisted entity, a null reference will be returned.

If the property extraction operator (.) is used to access a property of an entity, the reference is automatically dereferenced.

Example

The following Entity SQL query uses the REF operator to return the reference for an input entity argument. The same query dereferences the reference because we are using a property extraction operation (.) to access a property of the Product entity. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns PrimitiveType Results.
2. Pass the following query as an argument to the `ExecutePrimitiveTypeQuery` method:

```
SELECT VALUE REF(p).Name FROM AdventureWorksEntities.Products AS p
```

See also

- DEREF
- CREATEREF
- KEY
- Entity SQL Reference
- Type Definitions

ROW (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Constructs anonymous, structurally typed records from one or more values.

Syntax

```
ROW ( expression [ AS alias ] [,...] )
```

Arguments

`expression` Any valid query expression that returns a value to construct in a row type.

`alias` Specifies an alias for the value specified in a row type. If an alias is not provided, Entity SQL tries to generate an alias based on the Entity SQL alias generation rules.

Return Value

A row type.

Remarks

You use row constructors in the Entity SQL to construct anonymous, structurally typed records from one or more values. The result type of a row constructor is a row type whose field types correspond to the types of the values that were used to construct the row. For example, the following expression constructs a value of type `Record(a int, b string, c int)`.

```
ROW(1 AS a, "abc" AS b, a+34 AS c)
```

If you do not provide an alias for an expression in a row constructor, the Entity Framework will try to generate one. For more information, see the "Aliasing Rules" section of the Identifiers topic.

The following rules apply to expression aliasing in a row constructor:

- Expressions in a row constructor cannot refer to other aliases in the same constructor.
- Two expressions in the same row constructor cannot have the same alias.

For more information about query constructors, see Constructing Types.

Example

The following Entity SQL query uses the ROW operator to construct anonymous, structurally typed records. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE ROW (product.ProductID AS ProductID,
    product.Name AS ProductName) FROM AdventureWorksEntities.Products
    AS product
```

See also

- Constructing Types
- Entity SQL Reference
- Type Definitions

SELECT (Entity SQL)

11/8/2022 • 3 minutes to read • Edit Online

Specifies the elements returned by a query.

Syntax

```
SELECT [ ALL | DISTINCT ] [ topSubclause ] aliasedExpr
      [{ , aliasedExpr }] FROM fromClause [ WHERE whereClause ] [ GROUP BY groupByClause [ HAVING
havingClause ] ] [ ORDER BY orderByClause ]
-- or
SELECT VALUE [ ALL | DISTINCT ] [ topSubclause ] expr FROM fromClause [ WHERE whereClause ] [ GROUP BY
groupByClause [ HAVING havingClause ] ] [ ORDER BY orderByClause
```

Arguments

ALL Specifies that duplicates can appear in the result set. ALL is the default.

DISTINCT Specifies that only unique results can appear in the result set.

VALUE Allows only one item to be specified, and does not add on a row wrapper.

`topSubclause` Any valid expression that indicates the number of first results to return from the query, of the form `top(expr)`.

The LIMIT parameter of the ORDER BY operator also lets you select the first n items in the result set.

`aliasedExpr` An expression of the form:

`expr` as `identifier` | `expr`

`expr` A literal or expression.

Remarks

The SELECT clause is evaluated after the FROM, GROUP BY, and HAVING clauses have been evaluated. The SELECT clause can only refer to items currently in-scope (from the FROM clause, or from outer scopes). If a GROUP BY clause has been specified, the SELECT clause is only allowed to reference the aliases for the GROUP BY keys. Referring to the FROM clause items is only permitted in aggregate functions.

The list of one or more query expressions following the SELECT keyword is known as the select list, or more formally as the projection. The most general form of projection is a single query expression. If you select a member `member1` from a collection `collection1`, you will produce a new collection of all the `member1` values for each object in `collection1`, as illustrated in the following example.

```
SELECT collection1.member1 FROM collection1
```

For example, if `customers` is a collection of type `Customer` that has a property `Name` that is of type `string`, selecting `Name` from `customers` will yield a collection of strings, as illustrated in the following example.

```
SELECT customers.Name FROM customers AS c
```

It is also possible to use JOIN syntax (FULL, INNER, LEFT, OUTER, ON, and RIGHT). ON is required for inner joins and is nto allowed for cross joins.

Row and Value Select Clauses

Entity SQL supports two variants of the SELECT clause. The first variant, row select, is identified by the SELECT keyword, and can be used to specify one or more values that should be projected out. Because a row wrapper is implicitly added around the values returned, the result of the query expression is always a multiset of rows.

Each query expression in a row select must specify an alias. If no alias is specified,Entity SQL attempts to generate an alias by using the alias generation rules.

The other variant of the SELECT clause, value select, is identified by the SELECT VALUE keyword. It allows only one value to be specified, and does not add a row wrapper.

A row select is always expressible in terms of VALUE SELECT, as illustrated in the following example.

```
SELECT 1 AS a, "abc" AS b FROM C
SELECT VALUE ROW(1 AS a, "abc" AS b) FROM C
```

All and Distinct Modifiers

Both variants of SELECT in Entity SQL allow the specification of an ALL or DISTINCT modifier. If the DISTINCT modifier is specified, duplicates are eliminated from the collection produced by the query expression (up to and including the SELECT clause). If the ALL modifier is specified, no duplicate elimination is performed; ALL is the default.

Differences from Transact-SQL

Unlike Transact-SQL, Entity SQL does not support use of the * argument in the SELECT clause. Instead, Entity SQL allows queries to project out entire records by referencing the collection aliases from the FROM clause, as illustrated in the following example.

```
SELECT * FROM T1, T2
```

The previous Transact-SQL query expression is expressed in Entity SQL in the following way.

```
SELECT a1, a2 FROM T1 AS a1, T2 AS a2
```

Example

The following Entity SQL query uses the SELECT operator to specify the elements to be returned by a query. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE product FROM AdventureWorksEntities.Products
    AS product WHERE product.ListPrice < @price
```

See also

- Query Expressions
- Entity SQL Reference
- TOP

SET (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

The SET expression is used to convert a collection of objects into a set by yielding a new collection with all duplicate elements removed.

Syntax

```
SET ( expression )
```

Arguments

`expression` Any valid query expression that returns a collection.

Remarks

The set expression `SET(c)` is logically equivalent to the following select statement:

```
SELECT VALUE DISTINCT c FROM c
```

`SET` is one of the Entity SQL set operators. All Entity SQL set operators are evaluated from left to right. See EXCEPT for precedence information for the Entity SQL set operators.

Example

The following Entity SQL query uses the SET expression to convert a collection of objects into a set. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns PrimitiveType Results.
2. Pass the following query as an argument to the `ExecutePrimitiveTypeQuery` method:

```
SET(SELECT VALUE P.Name FROM AdventureWorksEntities.Products AS P)
```

See also

- Entity SQL Reference

SKIP (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

You can perform physical paging by using the SKIP sub-clause in the ORDER BY clause. SKIP cannot be used separately from the ORDER BY clause.

Syntax

```
[ SKIP n ]
```

Arguments

`n`
The number of items to skip.

Remarks

If a SKIP expression sub-clause is present in an ORDER BY clause, the results will be sorted according the sort specification and the result set will include rows starting from the next row immediately after the SKIP expression. For example, SKIP 5 will skip the first five rows and return from the sixth row forward.

> **NOTE**
> An Entity SQL query is invalid if both the TOP modifier and the SKIP sub-clause are present in the same query expression. The query should be rewritten by changing the TOP expression to the LIMIT expression.

> **NOTE**
> In SQL Server 2000, using SKIP with ORDER BY on non-key columns might return incorrect results. More than the specified number of rows might be skipped if the non-key column has duplicate data in it. This is due to how SKIP is translated for SQL Server 2000. For example, in the following code more than five rows might be skipped if `E.NonKeyColumn` has duplicate values:
>
> ```
> SELECT [E] FROM Container.EntitySet AS [E] ORDER BY [E].[NonKeyColumn] DESC SKIP 5L
> ```

The Entity SQL query in How to: Page Through Query Results uses the ORDER BY operator with SKIP to specify the sort order used on objects returned in a SELECT statement.

See also

- ORDER BY
- How to: Page Through Query Results
- Paging
- TOP

THEN (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

The result of a WHEN clause when it evaluates to `true`.

Syntax

```
WHEN when_expression THEN then_expression
```

Arguments

`when_expression`
Any valid Boolean expression.

`then_expression`
Any valid query expression that returns a collection.

Remarks

If `when_expression` evaluates to the value `true`, the result is the corresponding `then-expression`. If none of the WHEN conditions are satisfied, the `else-expression` is evaluated. However, if there is no `else-expression`, the result is null.

For an example, see CASE.

Example

The following Entity SQL query uses the CASE expression to evaluate a set of `Boolean` expressions. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns PrimitiveType Results.
2. Pass the following query as an argument to the `ExecutePrimitiveTypeQuery` method:

```
CASE WHEN AVG({@score1,@score2,@score3}) < @total THEN TRUE ELSE FALSE END
```

See also

- CASE
- Entity SQL Reference

TOP (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

The SELECT clause can have an optional TOP sub-clause following the optional ALL/DISTINCT modifier. The TOP sub-clause specifies that only the first set of rows will be returned from the query result.

Syntax

```
[ TOP (n) ]
```

Arguments

`n` The numeric expression that specifies the number of rows to be returned. `n` could be a single numeric literal or a single parameter.

Remarks

The TOP expression must be either a single numeric literal or a single parameter. If a constant literal is used, the literal type must be implicitly promotable to Edm.Int64 (byte, int16, int32 or int64 or any provider type that maps to a type that is promotable to Edm.Int64) and its value must be greater than or equal to zero. Otherwise an exception will be raised. If a parameter is used as an expression, the parameter type must also be implicitly promotable to Edm.Int64, but there will be no validation of the actual parameter value during compilation because the parameter values are late bounded.

The following is an example of constant TOP expression:

```
select distinct top(10) c.a1, c.a2 from T as a
```

The following is an example of parameterized TOP expression:

```
select distinct top(@topParam) c.a1, c.a2 from T as a
```

TOP is non-deterministic unless the query is sorted. If you require a deterministic result, use the SKIP and LIMIT sub-clauses in the ORDER BY clause. The TOP and SKIP/LIMIT are mutually exclusive.

Example

The following Entity SQL query uses the TOP to specify the top one row to be returned from the query result. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
SELECT VALUE TOP(1) contact FROM AdventureWorksEntities.Contacts AS contact
```

See also

- SELECT
- SKIP
- LIMIT
- ORDER BY
- Entity SQL Reference

TREAT (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Treats an object of a particular base type as an object of the specified derived type.

Syntax

```
TREAT ( expression as type)
```

Arguments

`expression` Any valid query expression that returns an entity.

> **NOTE**
> The type of the specified expression must be a subtype of the specified data type, or the data type must be a subtype of the type of expression.

`type` An entity type. The type must be qualified by a namespace.

> **NOTE**
> The specified expression must be a subtype of the specified data type, or the data type must be a subtype of the expression.

Return Value

A value of the specified data type.

Remarks

TREAT is used to perform upcasting between related classes. For example, if `Employee` derives from `Person` and p is of type `Person`, `TREAT(p AS NamespaceName.Employee)` upcasts a generic `Person` instance to `Employee`; that is, it allows you to treat p as `Employee`.

TREAT is used in inheritance scenarios where you can do a query like the following:

```
SELECT TREAT(p AS NamespaceName.Employee)
FROM ContainerName.Person AS p
WHERE p IS OF (NamespaceName.Employee)
```

This query upcasts `Person` entities to the `Employee` type. If the value of p is not actually of type `Employee`, the expression yields the value `null`.

NOTE

The specified expression `Employee` must be a subtype of the specified data type `Person`, or the data type must be a subtype of the expression. Otherwise, the expression will result in a compile-time error.

The following table shows the behavior of treat over some typical patterns and some less common patterns. All exceptions are thrown from the client side before the provider gets invoked:

PATTERN	BEHAVIOR
`TREAT (null AS EntityType)`	Returns `DbNull`.
`TREAT (null AS ComplexType)`	Throws an exception.
`TREAT (null AS RowType)`	Throws an exception/
`TREAT (EntityType AS EntityType)`	Returns `EntityType` or `null`.
`TREAT (ComplexType AS ComplexType)`	Throws an exception.
`TREAT (RowType AS RowType)`	Throws an exception.

Example

The following Entity SQL query uses the TREAT operator to convert an object of the type Course to a collection of objects of the type OnsiteCourse. The query is based on the School Model.

```
SELECT VALUE TREAT (course AS SchoolModel.OnsiteCourse)
   FROM SchoolEntities.Courses AS course
   WHERE course IS OF( SchoolModel.OnsiteCourse)
```

See also

- Entity SQL Reference
- Nullable Structured Types

UNION (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Combines the results of two or more queries into a single collection.

Syntax

```
expression
UNION [ ALL ]
expression
```

Arguments

`expression` Any valid query expression that returns a collection to combine with the collection All expressions must be of the same type or of a common base or derived type as `expression`.

UNION Specifies that multiple collections are to be combined and returned as a single collection.

ALL Specifies that multiple collections are to be combined and returned as a single collection, including duplicates. If not specified, duplicates are removed from the result collection.

Return Value

A collection of the same type or of a common base or derived type as `expression`.

Remarks

UNION is one of the Entity SQL set operators. All Entity SQL set operators are evaluated from left to right. For precedence information for the Entity SQL set operators, see EXCEPT.

Example

The following Entity SQL query uses the UNION ALL operator to combine the results of two queries into a single collection. The query is based on the AdventureWorks Sales Model. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns StructuralType Results.
2. Pass the following query as an argument to the `ExecuteStructuralTypeQuery` method:

```
(SELECT VALUE P FROM AdventureWorksEntities.Products
    AS P WHERE P.Name LIKE 'C%') UNION ALL
    (SELECT VALUE A FROM AdventureWorksEntities.Products
    AS A WHERE A.ListPrice > @price)
```

See also

- Entity SQL Reference

USING (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

Specifies namespaces used in a query expression.

Syntax

```
USING [ alias = ] namespace
```

Arguments

`alias`
Specifies a shorter alias to qualify a namespace with.

`namespace`
Any valid namespace.

Example

The following Entity SQL query uses the USING operator to specify namespaces used in a query expression. To compile and run this query, follow these steps:

1. Follow the procedure in How to: Execute a Query that Returns PrimitiveType Results.
2. Pass the following query as an argument to the `ExecutePrimitiveTypeQuery` method:

```
using SqlServer; RAND()
```

See also

- Namespaces
- Entity SQL Reference

WHERE (Entity SQL)

11/8/2022 • 2 minutes to read • Edit Online

The WHERE clause is applied directly after the FROM clause.

Syntax

```
[ WHERE expression ]
```

Arguments

`expression`
A Boolean type.

Remarks

The WHERE clause has the same semantics as described for Transact-SQL. It restricts the objects produced by the query expression by limiting the elements of the source collections to those that pass the condition.

```
select c from cs as c where e
```

The expression `e` must have the type Boolean.

The WHERE clause is applied directly after the FROM clause and before any grouping, ordering, or projection takes place. All element names defined in the FROM clause are visible to the expression of the WHERE clause.

See also

- Entity SQL Reference
- Query Expressions

Canonical Functions

11/8/2022 • 2 minutes to read • Edit Online

This section discusses canonical functions that are supported by all data providers, and can be used by all querying technologies. Canonical functions cannot be extended by a provider.

These canonical functions will be translated to the corresponding data source functionality for the provider. This allows for function invocations expressed in a common form across data sources.

Because these canonical functions are independent of data sources, argument and return types of canonical functions are defined in terms of types in the conceptual model. However, some data sources might not support all types in the conceptual model.

When canonical functions are used in an Entity SQL query, the appropriate function will be called at the data source.

All canonical functions have both null-input behavior and error conditions explicitly specified. Store providers should comply with that behavior, but Entity Framework does not enforce this behavior.

For LINQ scenarios, queries against the Entity Framework involve mapping CLR methods to methods in the underlying data source. The CLR methods map to canonical functions, so that a specific set of methods will correctly map, regardless of the data source.

Canonical Functions Namespace

The namespace for canonical function is System.Data.Metadata.Edm. The System.Data.Metadata.Edm namespace is automatically included in all queries. However, if another namespace is imported that contains a function with the same name as a canonical function (in the System.Data.Metadata.Edm namespace), you must specify the namespace.

In This Section

Aggregate Canonical Functions Discusses aggregate Entity SQL canonical functions.

Math Canonical Functions Discusses math Entity SQL canonical functions.

String Canonical Functions Discusses string Entity SQL canonical functions.

Date and Time Canonical Functions Discusses date and time Entity SQL canonical functions.

Bitwise Canonical Functions Discusses bitwise Entity SQL canonical functions.

Spatial Functions Discusses Spatial Entity SQL canonical functions.

Other Canonical Functions Discusses functions not classified as bitwise, date/time, string, math, or aggregate.

See also

- Entity SQL Overview
- Entity SQL Reference
- Conceptual Model Canonical to SQL Server Functions Mapping
- User-Defined Functions

Aggregate Canonical Functions

11/8/2022 • 4 minutes to read • Edit Online

Aggregates are expressions that reduce a series of input values into, for example, a single value. Aggregates are normally used in conjunction with the GROUP BY clause of the SELECT expression, and there are constraints on where they can be used.

Aggregate Entity SQL canonical functions

The following are the aggregate Entity SQL canonical functions.

Avg(expression)

Returns the average of the non-null values.

Arguments

An `Int32`, `Int64`, `Double`, and `Decimal`.

Return Value

The type of `expression`, or `null` if all input values are `null` values.

Example

```
queryString = @"SELECT VALUE AVG(p.ListPrice)
    FROM AdventureWorksEntities.Products as p";
```

```
SELECT VALUE AVG(p.ListPrice)
FROM AdventureWorksEntities.Products AS p
```

BigCount(expression)

Returns the size of the aggregate including null and duplicate values.

Arguments

Any type.

Return Value

An `Int64`.

Example

```
queryString = @"SELECT VALUE BigCount(p.ProductID)
    FROM AdventureWorksEntities.Products as p";
```

```
SELECT VALUE BigCount(p.ProductID)
FROM AdventureWorksEntities.Products AS p
```

Count(expression)

Returns the size of the aggregate including null and duplicate values.

Arguments

Any type.

Return Value

An `Int32`.

Example

```
queryString = @"SELECT VALUE Count(p.ProductID)
    FROM AdventureWorksEntities.Products as p";
```

```
SELECT VALUE Count(p.ProductID)
FROM AdventureWorksEntities.Products AS p
```

Max(expression)

Returns the maximum of the non-null values.

Arguments

A `Byte`, `Int16`, `Int32`, `Int64`, `Byte`, `Single`, `Double`, `Decimal`, `DateTime`, `DateTimeOffset`, `Time`, `String`, `Binary`.

Return Value

The type of `expression`, or `null` if all input values are `null` values.

Example

```
queryString = @"SELECT VALUE MAX(p.ListPrice)
    FROM AdventureWorksEntities.Products as p";
```

```
SELECT VALUE MAX(p.ListPrice)
FROM AdventureWorksEntities.Products AS p
```

Min(expression)

Returns the minimum of the non-null values.

Arguments

A `Byte`, `Int16`, `Int32`, `Int64`, `Byte`, `Single`, `Double`, `Decimal`, `DateTime`, `DateTimeOffset`, `Time`, `String`, `Binary`.

Return Value

The type of `expression`, or `null` if all input values are `null` values.

Example

```
queryString = @"SELECT VALUE MIN(p.ListPrice)
    FROM AdventureWorksEntities.Products as p";
```

```
SELECT VALUE MIN(p.ListPrice)
FROM AdventureWorksEntities.Products AS p
```

StDev(expression)

Returns the standard deviation of the non-null values.

Arguments

An `Int32`, `Int64`, `Double`, `Decimal`.

Return Value

A `Double`. `Null`, if all input values are `null` values.

Example

```
queryString = @"SELECT VALUE StDev(product.ListPrice)
    FROM AdventureWorksEntities.Products AS product
    WHERE product.ListPrice > @price";
```

```
SELECT VALUE StDev(product.ListPrice)
FROM AdventureWorksEntities.Products AS product
WHERE product.ListPrice > @price
```

StDevP(expression)

Returns the standard deviation for the population of all values.

Arguments

An `Int32`, `Int64`, `Double`, `Decimal`.

Return Value

A `Double`, or `null` if all input values are `null` values.

Example

```
queryString = @"SELECT VALUE StDevP(product.ListPrice)
    FROM AdventureWorksEntities.Products AS product
    WHERE product.ListPrice > @price";
```

```
SELECT VALUE StDevP(product.ListPrice)
FROM AdventureWorksEntities.Products AS product
WHERE product.ListPrice > @price
```

Sum(expression)

Returns the sum of the non-null values.

Arguments

An `Int32`, `Int64`, `Double`, `Decimal`.

Return Value

A `Double`, or `null` if all input values are `null` values.

Example

```
queryString = @"SELECT VALUE Sum(p.ListPrice)
    FROM AdventureWorksEntities.Products as p";
```

```
SELECT VALUE Sum(p.ListPrice)
FROM AdventureWorksEntities.Products AS p
```

Var(expression)

Returns the variance of all non-null values.

Arguments

An `Int32`, `Int64`, `Double`, `Decimal`.

Return Value

A `Double`, or `null` if all input values are `null` values.

Example

```
queryString = @"SELECT VALUE Var(product.ListPrice)
    FROM AdventureWorksEntities.Products AS product
    WHERE product.ListPrice > @price";
```

```
SELECT VALUE Var(product.ListPrice)
FROM AdventureWorksEntities.Products AS product
WHERE product.ListPrice > @price
```

VarP(expression)

Returns the variance for the population of all non-null values.

Arguments

An `Int32`, `Int64`, `Double`, `Decimal`.

Return Value

A `Double`, or `null` if all input values are `null` values.

Example

```
queryString = @"SELECT VALUE VarP(product.ListPrice)
    FROM AdventureWorksEntities.Products AS product
    WHERE product.ListPrice > @price";
```

```
SELECT VALUE VarP(product.ListPrice)
FROM AdventureWorksEntities.Products AS product
WHERE product.ListPrice > @price
```

Equivalent functionality is available in the Microsoft SQL Client Managed Provider. For more information, see SqlClient for Entity Framework Functions.

Collection-based aggregates

Collection-based aggregates (collection functions) operate on collections and return a value. For example if ORDERS is a collection of all orders, you can calculate the earliest ship date with the following expression:

```
min(select value o.ShipDate from LOB.Orders as o)
```

Expressions inside collection-based aggregates are evaluated within the current ambient name-resolution scope.

Group-based aggregates

Group-based aggregates are calculated over a group as defined by the GROUP BY clause. For each group in the result, a separate aggregate is calculated by using the elements in each group as input to the aggregate calculation. When a group-by clause is used in a select expression, only grouping expression names, aggregates, or constant expressions can be present in the projection or order-by clause.

The following example calculates the average quantity ordered for each product:

```
select p, avg(ol.Quantity) from LOB.OrderLines as ol
  group by ol.Product as p
```

It's possible to have a group-based aggregate without an explicit group-by clause in the SELECT expression. In this case, all elements are treated as a single group. This is equivalent of specifying a grouping based on a constant. Take, for example, the following expression:

```
select avg(ol.Quantity) from LOB.OrderLines as ol
```

This is equivalent to the following:

```
select avg(ol.Quantity) from LOB.OrderLines as ol group by 1
```

Expressions inside the group-based aggregate are evaluated within the name-resolution scope that would be visible to the WHERE clause expression.

As in Transact-SQL, group-based aggregates can also specify an ALL or DISTINCT modifier. If the DISTINCT modifier is specified, duplicates are eliminated from the aggregate input collection, before the aggregate is computed. If the ALL modifier is specified (or if no modifier is specified), no duplicate elimination is performed.

See also

- Canonical Functions

Math Canonical Functions

11/8/2022 • 2 minutes to read • Edit Online

Entity SQL includes the following math canonical functions:

Abs(value)

Returns the absolute value of `value`.

Arguments

An `Int16`, `Int32`, `Int64`, `Byte`, `Single`, `Double`, and `Decimal`.

Return Value

The type of `value`.

Example

`Abs(-2)`

Ceiling(value)

Returns the smallest integer that is not less than `value`.

Arguments

A `Single`, `Double`, and `Decimal`.

Return Value

The type of `value`.

Example

```
SELECT VALUE product FROM AdventureWorksEntities.Products AS product
WHERE CEILING(product.ListPrice) == FLOOR(product.ListPrice)
```


```
SELECT VALUE product FROM AdventureWorksEntities.Products AS product
WHERE CEILING(product.ListPrice) == FLOOR(product.ListPrice)
```


Floor(value)

Returns the largest integer that is not greater than `value`.

Arguments

A `Single`, `Double`, and `Decimal`.

Return Value

The type of `value`.

Example

```
SELECT VALUE product FROM AdventureWorksEntities.Products AS product
WHERE FLOOR(product.ListPrice) == CEILING(product.ListPrice)
```


```
SELECT VALUE product FROM AdventureWorksEntities.Products AS product
WHERE FLOOR(product.ListPrice) == CEILING(product.ListPrice)
```


Power(value, exponent)

Returns the result of the specified `value` to the specified `exponent`.

Arguments

PARAMETER	TYPE
`value`	`Int32`, `Int64`, `Double`, or `Decimal`.
`exponent`	`Int64`, `Double`, or `Decimal`.

Return Value

The type of `value`.

Example

`Power(748.58,2)`

Round(value)

Returns the integer portion of `value`, rounded to the nearest integer.

Arguments

A `Single`, `Double`, and `Decimal`.

Return Value

The type of `value`.

Example

`Round(748.58)`

Round(value, digits)

Returns the `value`, rounded to the nearest specified `digits`.

Arguments

PARAMETER	TYPE
`value`	`Double` or `Decimal`.
`digits`	`Int16` or `Int32`.

Return Value

The type of `value`.

Example

```
Round(748.58,1)
```

Truncate(value, digits)

Returns the `value`, truncated to the nearest specified `digits`.

Arguments

PARAMETER	TYPE
`value`	`Double` or `Decimal`.
`digits`	`Int16` or `Int32`.

Return Value

The type of `value`.

Example

```
Truncate(748.58,1)
```

These functions will return `null` if given `null` input.

Equivalent functionality is available in the Microsoft SQL Client Managed Provider. For more information, see SqlClient for Entity Framework Functions.

See also

- Canonical Functions

String Canonical Functions

11/8/2022 • 3 minutes to read • Edit Online

Entity SQL includes string canonical functions.

Remarks

The following table shows the string Entity SQL canonical functions.

FUNCTION	DESCRIPTION

`Concat(string1, string2)`	Returns a string that contains `string2` appended to `string1`.
	
	Arguments
	
	`string1`: The string to which `string2` is appended.
	
	`string2`: The string that is appended to `string1`.
	
	Return Value
	
	A `String`. An error will occur if the length of the return value string is greater than the maximum length allowed.
	
	Example
	
	`-- The following example returns abcxyz.`
	
	`Concat('abc', 'xyz')`

`Contains(string, target)`	Returns `true` if `target` is contained in `string`.
	
	Arguments
	
	`string`: The string that is searched.
	
	`target`: The target string that is searched for.
	
	Return Value
	
	`true` if `target` is contained in `string`; otherwise `false`.
	
	Example
	
	`-- The following example returns true.`
	
	`Contains('abc', 'bc')`

FUNCTION	DESCRIPTION

`EndsWith(string, target)`	Returns `true` if `target` ends with `string`.
	
	Arguments
	
	`string`: The string that is searched.
	
	`target`: The target string searched for at the end of `string`.
	
	Return Value
	
	`True` if `string` ends with `target`; otherwise `false`.
	
	Example
	
	`-- The following example returns true.`
	
	`EndsWith('abc', 'bc')` **Note:** If you are using the SQL Server data provider, this function returns `false` if the string is stored in a fixed length string column and `target` is a constant. In this case, the entire string is searched, including any padding trailing spaces. A possible workaround is to trim the data in the fixed length string, as in the following example: `EndsWith(TRIM(string), target)`

`IndexOf(target, string)`	Returns the position of `target` inside `string`, or 0 if not found. Returns 1 to indicate the beginning of `string`. Index numbering starts from 1.
	
	Arguments
	
	`target`: The string that is searched for.
	
	`string`: The string that is searched.
	
	Return Value
	
	An `Int32`.
	
	Example
	
	`-- The following example returns 4.`
	
	`IndexOf('xyz', 'abcxyz')`

FUNCTION	DESCRIPTION

`Left(string, length)`	Returns the first `length` characters from the left side of `string`. If the length of `string` is less than `length`, the entire string is returned.
	
	Arguments
	
	`string`: A `String`.
	
	`length`: An `Int16`, `Int32`, `Int64`, or `Byte`. `length` cannot be less than zero.
	
	Return Value
	
	A `String`.
	
	Example
	
	`-- The following example returns abc.`
	
	`Left('abcxyz', 3)`

`Length(string)`	Returns the (`Int32`) length, in characters, of the string.
	
	Arguments
	
	`string`: A `String`.
	
	Return Value
	
	An `Int32`.
	
	Example
	
	`-- The following example returns 6.`
	
	`Length('abcxyz')`

`LTrim(string)`	Returns `string` without leading white space.
	
	Arguments
	
	A `String`.
	
	Return Value
	
	A `String`.
	
	Example
	
	`-- The following example returns abc.`
	
	`LTrim(' abc')`

FUNCTION	DESCRIPTION

`Replace(string1, string2, string3)`	Returns `string1`, with all occurrences of `string2` replaced by `string3`.
	
	Arguments
	
	A `String`.
	
	Return Value
	
	A `String`.
	
	Example
	
	`-- The following example returns abcxyz.`
	
	`Concat('abc', 'xyz')`

`Reverse(string)`	Returns `string` with the order of the characters reversed.
	
	Arguments
	
	A `String`.
	
	Return Value
	
	A `String`.
	
	Example
	
	`-- The following example returns dcba.`
	
	`Reverse('abcd')`

`Right(string, length)`	Returns the last `length` characters from the `string`. If the length of `string` is less than `length`, the entire string is returned.
	
	Arguments
	
	`string`: A `String`.
	
	`length`: An `Int16`, `Int32`, `Int64`, or `Byte`. `length` cannot be less than zero.
	
	Return Value
	
	A `String`.
	
	Example
	
	`-- The following example returns xyz.`
	
	`Right('abcxyz', 3)`

FUNCTION	DESCRIPTION
`RTrim(string)`	Returns `string` without trailing white space. **Arguments** A `String`. **Return Value** A `String`.

`Substring(string, start, length)`	Returns the substring of the string starting at position `start`, with a length of `length` characters. A start of 1 indicates the first character of the string. Index numbering starts from 1.
	
	Arguments
	
	`string`: A `String`.
	
	`start`: An `Int16`, `Int32`, `Int64` and `Byte`. `start` cannot be less than one.
	
	`length`: An `Int16`, `Int32`, `Int64` and `Byte`. `length` cannot be less than zero.
	
	Return Value
	
	A `String`.
	
	Example
	
	`-- The following example returns xyz.`
	
	`Substring('abcxyz', 4, 3)`

`StartsWith(string, target)`	Returns `true` if `string` starts with `target`.
	
	Arguments
	
	`string`: The string that is searched.
	
	`target`: The target string searched for at the start of `string`.
	
	Return Value
	
	`True` if `string` starts with `target`; otherwise `false`.
	
	Example
	
	`-- The following example returns true.`
	
	`StartsWith('abc', 'ab')`

FUNCTION	DESCRIPTION

`ToLower(string)`	Returns `string` with uppercase characters converted to lowercase.
	
	Arguments
	
	A `String`.
	
	Return Value
	
	A `String`.
	
	Example
	
	`-- The following example returns abc.`
	
	`ToLower('ABC')`

`ToUpper(string)`	Returns `string` with lowercase characters converted to uppercase.
	
	Arguments
	
	A `String`.
	
	Return Value
	
	A `String`.
	
	Example
	
	`-- The following example returns ABC.`
	
	`ToUpper('abc')`

`Trim(string)`	Returns `string` without leading and trailing white space.
	
	Arguments
	
	A `String`.
	
	Return Value
	
	A `String`.
	
	Example
	
	`-- The following example returns abc.`
	
	`Trim(' abc ')`

These functions will return `null` if given `null` input.

Equivalent functionality is available in the Microsoft SQL Client Managed Provider. For more information, see SqlClient for Entity Framework Functions.

See also

- Canonical Functions

Date and Time Canonical Functions

11/8/2022 • 5 minutes to read • Edit Online

Entity SQL includes date and time canonical functions.

Remarks

The following table shows the date and time Entity SQL canonical functions. `datetime` is a DateTime value.

FUNCTION	DESCRIPTION
`AddNanoseconds(expression,number)`	Adds the specified `number` of nanoseconds to the `expression`. **Arguments** `expression`: `DateTime`, `DateTimeOffset`, or `Time`. `number`: `Int32`. **Return Value** The type of `expression`.
`AddMicroseconds(expression,number)`	Adds the specified `number` of microseconds to the `expression`. **Arguments** `expression`: `DateTime`, `DateTimeOffset`, or `Time`. `number`: `Int32`. **Return Value** The type of `expression`.
`AddMilliseconds(expression,number)`	Adds the specified `number` of milliseconds to the `expression`. **Arguments** `expression`: `DateTime`, `DateTimeOffset`, or `Time`. `number`: `Int32`. **Return Value** The type of `expression`.

FUNCTION	DESCRIPTION
`AddSeconds(expression,number)`	Adds the specified `number` of seconds to the `expression`. **Arguments** `expression`: `DateTime`, `DateTimeOffset`, or `Time`. `number`: `Int32`. **Return Value** The type of `expression`.
`AddMinutes(expression,number)`	Adds the specified `number` of minutes to the `expression`. **Arguments** `expression`: `DateTime`, `DateTimeOffset`, or `Time`. `number`: `Int32`. **Return Value** The type of `expression`.
`AddHours(expression,number)`	Adds the specified `number` of hours to the `expression`. **Arguments** `expression`: `DateTime`, `DateTimeOffset`, or `Time`. `number`: `Int32`. **Return Value** The type of `expression`.
`AddDays(expression,number)`	Adds the specified `number` of days to the `expression`. **Arguments** `expression`: `DateTime` or `DateTimeOffset`. `number`: `Int32`. **Return Value** The type of `expression`.

FUNCTION	DESCRIPTION
`AddMonths(expression,number)`	Adds the specified `number` of months to the `expression`. **Arguments** `expression`: `DateTime` or `DateTimeOffset`. `number`: `Int32`. **Return Value** The type of `expression`.
`AddYears(expression,number)`	Adds the specified `number` of years to the `expression`. **Arguments** `expression`: `DateTime` or `DateTimeOffset`. `number`: `Int32`. **Return Value** The type of `expression`.
`CreateDateTime(year,month,day,hour,minute,second)`	Returns a new `DateTime` value as the current date and time of the server in the server's time zone. **Arguments** `year`, `month`, `day`, `hour`, `minute`: `Int16` and `Int32`. `second`: `Double`. **Return Value** A `DateTime`.
`CreateDateTimeOffset(year,month,day,hour,minute,second,tzoffset)`	Returns a new `DateTimeOffset` value as the current date and time of the server relative to the Coordinated Universal Time (UTC). **Arguments** `year`, `month`, `day`, `hour`, `minute`, `tzoffset`: `Int32`. `second`: `Double`. **Return Value** A `DateTimeOffset`.

FUNCTION	DESCRIPTION
`CreateTime(hour,minute,second)`	Returns a new `Time` value as the current time. **Arguments** `hour` and `minute`: `Int32`. `second`: `Double`. **Return Value** A `Time`.
`CurrentDateTime()`	Returns a `DateTime` value as the current date and time of the server in the server's time zone. **Return Value** A `DateTime`.
`CurrentDateTimeOffset()`	Returns the current date, time and offset as a `DateTimeOffset`. **Return Value** A `DateTimeOffset`.
`CurrentUtcDateTime()`	Returns a DateTime value as the current date and time of the server in the UTC time zone. **Return Value** A `DateTime`.

`Day(expression)`	Returns the day portion of `expression` as an `Int32` between 1 and 31.
	
	Arguments
	
	A `DateTime` and `DateTimeOffset`.
	
	Return Value
	
	An `Int32`.
	
	Example
	
	`-- The following example returns 12.`
	
	`Day(cast('03/12/1998' as DateTime))`

FUNCTION	DESCRIPTION
`DayOfYear(expression)`	Returns the day portion of `expression` as an `Int32` between 1 and 366, where 366 is returned for the last day of a leap year. **Arguments** A `DateTime` or `DateTimeOffset`. **Return Value** An `Int32`.
`DiffNanoseconds(startExpression,endExpression)`	Returns the difference, in nanoseconds, between `startExpression` and `endExpression`. **Arguments** `startExpression`, `endExpression`: `DateTime`, `DateTimeOffset`, or `Time`. **Note:** `startExpression` and `endExpression` must be of the same type. **Return Value** An `Int32`.
`DiffMilliseconds(startExpression,endExpression)`	Returns the difference, in milliseconds, between `startExpression` and `endExpression`. **Arguments** `startExpression`, `endExpression`: `DateTime`, `DateTimeOffset`, or `Time`. **Note:** `startExpression` and `endExpression` must be of the same type. **Return Value** An `Int32`.
`DiffMicroseconds(startExpression,endExpression)`	Returns the difference, in microseconds, between `startExpression` and `endExpression`. **Arguments** `startExpression`, `endExpression`: `DateTime`, `DateTimeOffset`, or `Time`. **Note:** `startExpression` and `endExpression` must be of the same type. **Return Value** An `Int32`.

FUNCTION	DESCRIPTION
`DiffSeconds(startExpression,endExpression)`	Returns the difference, in seconds, between `startExpression` and `endExpression`. **Arguments** `startExpression`, `endExpression`: `DateTime`, `DateTimeOffset`, or `Time`. **Note:** `startExpression` and `endExpression` must be of the same type. **Return Value** An `Int32`.
`DiffMinutes(startExpression,endExpression)`	Returns the difference, in minutes, between `startExpression` and `endExpression`. **Arguments** `startExpression`, `endExpression`: `DateTime`, `DateTimeOffset`, or `Time`. **Note:** `startExpression` and `endExpression` must be of the same type. **Return Value** An `Int32`.
`DiffHours(startExpression,endExpression)`	Returns the difference, in hours, between `startExpression` and `endExpression`. **Arguments** `startExpression`, `endExpression`: `DateTime`, `DateTimeOffset`, or `Time`. **Note:** `startExpression` and `endExpression` must be of the same type. **Return Value** An `Int32`.
`DiffDays(startExpression,endExpression)`	Returns the difference, in days, between `startExpression` and `endExpression`. **Arguments** `startExpression`, `endExpression`: `DateTime` or `DateTimeOffset`. **Note:** `startExpression` and `endExpression` must be of the same type. **Return Value** An `Int32`.

FUNCTION	DESCRIPTION
`DiffMonths(startExpression,endExpression)`	Returns the difference, in months, between `startExpression` and `endExpression`. **Arguments** `startExpression`, `endExpression`: `DateTime` or `DateTimeOffset`. **Note:** `startExpression` and `endExpression` must be of the same type. **Return Value** An `Int32`.
`DiffYears(startExpression,endExpression)`	Returns the difference, in years, between `startExpression` and `endExpression`. **Arguments** `startExpression`, `endExpression`: `DateTime` or `DateTimeOffset`. **Note:** `startExpression` and `endExpression` must be of the same type. **Return Value** An `Int32`.
`GetTotalOffsetMinutes(datetimeoffset)`	Returns the number of minutes that the `datetimeoffset` is offset from GMT. This is generally between +780 and -780 (+ or - 13 hrs). **Note:** This function is supported in SQL Server 2008 only. **Arguments** A `DateTimeOffset`. **Return Value** An `Int32`.

`Hour(expression)`	Returns the hour portion of `expression` as an `Int32` between 0 and 23.
	
	Arguments
	
	A `DateTime, Time` and `DateTimeOffset`.
	
	Example
	
	`-- The following example returns 22.`
	
	`Hour(cast('22:35:5' as DateTime))`

FUNCTION	DESCRIPTION
`Millisecond(expression)`	Returns the milliseconds portion of `expression` as an `Int32` between 0 and 999. **Arguments** A `DateTime, Time` and `DateTimeOffset`. **Return Value** An `Int32`.

`Minute(expression)`	Returns the minute portion of `expression` as an `Int32` between 0 and 59.
	
	Arguments
	
	A `DateTime, Time` or `DateTimeOffset`.
	
	Return Value
	
	An `Int32`.
	
	Example
	
	`-- The following example returns 35`
	
	`Minute(cast('22:35:5' as DateTime))`

`Month(expression)`	Returns the month portion of `expression` as an `Int32` between 1 and 12.
	
	Arguments
	
	A `DateTime` or `DateTimeOffset`.
	
	Return Value
	
	An `Int32`.
	
	Example
	
	`-- The following example returns 3.`
	
	`Month(cast('03/12/1998' as DateTime))`

FUNCTION	DESCRIPTION

`Second(expression)`	Returns the seconds portion of `expression` as an `Int32` between 0 and 59.
	
	Arguments
	
	A `DateTime, Time` and `DateTimeOffset`.
	
	Return Value
	
	An `Int32`.
	
	Example
	
	`-- The following example returns 5`
	
	`Second(cast('22:35:5' as DateTime))`

`TruncateTime(expression)`	Returns the `expression`, with the time values truncated. **Arguments** A `DateTime` or `DateTimeOffset`. **Return Value** The type of `expression`.

`Year(expression)`	Returns the year portion of `expression` as an `Int32` `YYYY`.
	
	Arguments
	
	A `DateTime` and `DateTimeOffset`.
	
	Return Value
	
	An `Int32`.
	
	Example
	
	`-- The following example returns 1998.`
	
	`Year(cast('03/12/1998' as DateTime))`

These functions will return `null` if given `null` input.

Equivalent functionality is available in the Microsoft SQL Client Managed Provider. For more information, see SqlClient for Entity Framework Functions.

See also

- Canonical Functions

Bitwise Canonical Functions

11/8/2022 • 2 minutes to read • Edit Online

Entity SQL includes bitwise canonical functions.

Remarks

The following table shows the bitwise Entity SQL canonical functions. These functions will return `Null` if `Null` input is provided. The return type of the functions is the same as the argument type(s). Arguments must be of the same type, if the function takes more than one argument. To perform bitwise operations across different types, you need to cast to the same type explicitly.

FUNCTION	DESCRIPTION

`BitWiseAnd ( value1 , value2 )`	Returns the bitwise conjunction of `value1` and `value2` as the type of `value1` and `value2`.
	
	Arguments
	
	A `Byte`, `Int16`, `Int32`, and `Int64`.
	
	Example
	
	`-- The following example returns 1.`
	
	`BitWiseAnd(1,3)`

`BitWiseNot ( value )`	Returns the bitwise negation of `value`.
	
	Arguments
	
	A `Byte`, `Int16`, `Int32`, and `Int64`.
	
	Example
	
	`-- The following example returns -4.`
	
	`BitWiseNot(3)`

`BitWiseOr ( value1 , value2 )`	Returns the bitwise disjunction of `value1` and `value2` as the type of `value1` and `value2`.
	
	Arguments
	
	A `Byte`, `Int16`, `Int32` and `Int64`.
	
	Example
	
	`-- The following example returns 3.`
	
	`BitWiseOr(1,3)`

FUNCTION	DESCRIPTION

`BitWiseXor ( value1 , value2 )`	Returns the bitwise exclusive disjunction of `value1` and `value2` as the type of `value1` and `value2`.
	
	Arguments
	
	A `Byte`, `Int16`, `Int32` and `Int64`.
	
	Example
	
	`-- The following example returns 2.`
	
	`BitWiseXor (1,3)`

See also

- Canonical Functions

Spatial Functions

11/8/2022 • 2 minutes to read • Edit Online

There is no literal format for spatial types. However, you can use canonical Entity Framework functions that you call using strings in Well-Known Text format. For example, the following function call creates a geometry point:

```
GeometryFromText('POINT (43 -73)')
```

The SpatialEdmFunctions methods have all spatial canonical Entity Framework methods. Click on a method of interest to see what parameters should be passed to a function.

Other Canonical Functions

11/8/2022 • 2 minutes to read • Edit Online

Entity SQL includes canonical functions not classified as bitwise, aggregate, math, date/time, or string.

The following table shows the other Entity SQL canonical functions.

FUNCTION	DESCRIPTION
`NewGuid()`	Returns a new GUID. **Example** `NewGuid()`

See also

- Canonical Functions