Contents

- Variables as parameters
- Parameterized SQL
  - Other parameterized queries
- Adhoc SQL
- Canonical functions

We can **pass parameters to a LINQ query** to control the SQL generated. Parameterized queries can result in improved performance: both in Entity Framework query plan compilation and in database execution plan caching and reuse.

Variables as parameters

Parameters to a LINQ query often take the form of local variables, but you can reference properties and fields too. When you pass a non-constant value into an EntityQuery, DevForce will "parameterize" the query, and ensure that the resulting SQL is a parameterized query.

For example, in the query below, we create a single query object but change the parameter *customerName* between executions of this query.

```
String customerName = null; // The customerName variable is a 'parameter'
var query = entityManager.Customers.Where(c => c.CompanyName.StartsWith(customerName));
customerName = "A";
var customersStartingWithA = query.ToList();
customerName = "B";
var customersStartingWithB = query.ToList();
```

```
Dim customerName As String = Nothing ' The customerName variable is a 'parameter'
Dim query = entityManager.Customers.Where(Function(c) c.CompanyName.StartsWith(customerName))
customerName = "A"
Dim customersStartingWithA= query.ToList()
customerName = "B"
Dim customersStartingWithB = query.ToList()
```

More complex queries that can be composed completely dynamically are discussed in **dynamic queries**.

One interesting side note is that the *EntityManager* treats parameterized and constant queries differently in its Query Cache. The following two queries are not equivalent:

```
// Parameterized query
customerName = "ABC";
var query1 = entityManager.Customers.Where(c => c.CompanyName.StartsWith(customerName));
var results1 = query1.ToList();
//Constant ("adhoc") query
var query2 = entityManager.Customers.Where(c => c.CompanyName.StartsWith("ABC"));
var results2 = query2.ToList();
```

Parameterized SQL

An *EntityQuery* using non-constant values, such as a local variable, field or property, will generate parameterized SQL in DevForce. This feature is available beginning in the 7.1.0 release.

For example, here's the query shown above which uses a local variable, *customerName*:

```
String customerName = "A";
var query = entityManager.Customers.Where(c => c.CompanyName.StartsWith(customerName));
var results = query.ToList();
```

It will generate the following parameterized SQL (in SQL Server):

```
exec sp_executesql N'SELECT
[Extent1].[CustomerID] AS [CustomerID],
[Extent1].[CustomerID_OLD] AS [CustomerID_OLD],
[Extent1].[CompanyName] AS [CompanyName],
[Extent1].[ContactName] AS [ContactName],
[Extent1].[ContactTitle] AS [ContactTitle],
[Extent1].[Address] AS [Address],
[Extent1].[City] AS [City],
[Extent1].[Region] AS [Region],
[Extent1].[PostalCode] AS [PostalCode],
[Extent1].[Country] AS [Country],
```

```
[Extent1].[Phone] AS [Phone],
[Extent1].[Fax] AS [Fax],
[Extent1].[RowVersion] AS [RowVersion]
FROM [dbo].[Customer] AS [Extent1]
WHERE [Extent1].[CompanyName] LIKE @p__linq__0 ESCAPE N'~',N'@p__linq__0 nvarchar(4000)',@p__linq__0=N'A%'
```

Using a variable in the *Where* clause indicates to DevForce that parameterized SQL is wanted. Parameterized SQL can provide improved database performance since the query execution plan can be reused with differing parameters. It can also provide improved Entity Framework performance, as EF will automatically "compile" the query plan once into its query plan cache, and reuse that cached plan.

Other parameterized queries

Lazily loaded navigation properties are considered parameterized queries, and generate parameterized SQL.

For example, retrieving all orders for a customer:

```
var orders = customer.Orders;
```

... results in the following SQL (in SQL Server):

```
exec sp_executesql N'SELECT
[Extent1].[OrderID] AS [OrderID],
[Extent1].[CustomerID] AS [CustomerID],
[Extent1].[EmployeeID] AS [EmployeeID],
[Extent1].[OrderDate] AS [OrderDate],
[Extent1].[RequiredDate] AS [RequiredDate],
[Extent1].[ShippedDate] AS [ShippedDate],
[Extent1].[Freight] AS [Freight],
[Extent1].[ShipName] AS [ShipName],
[Extent1].[ShipAddress] AS [ShipAddress],
[Extent1].[ShipCity] AS [ShipCity],
[Extent1].[ShipRegion] AS [ShipRegion],
[Extent1].[ShipPostalCode] AS [ShipPostalCode],
[Extent1].[ShipCountry] AS [ShipCountry],
[Extent1].[RowVersion] AS [RowVersion]
FROM [dbo].[Order] AS [Extent1]
WHERE ([Extent1].[CustomerID] IS NOT NULL) AND ([Extent1].[CustomerID] = @p__linq__0)',N'@p__linq__0
uniqueidentifier',@p__linq__0='729DE505-EA6D-4CDF-89F6-0360AD37BDE7'
```

The EntityKeyQuery also results in parameterized SQL, as does a PassthruEsqlQuery which uses QueryParameters.

Adhoc SQL

When using a profiler (such as SQL Server Profiler), you'll notice that some queries result in direct execution of a SQL statement. This occurs when the query does not contain any parameters.

For example, this query uses a hardcoded constant instead of a variable:

```
var query = entityManager.Customers.Where(c => c.CompanyName.StartsWith("A"));
var results = query.ToList();
```

The resulting SQL (in SQL Server) is an "adhoc" query:

```
SELECT
[Extent1].[CustomerID] AS [CustomerID],
[Extent1].[CustomerID_OLD] AS [CustomerID_OLD],
[Extent1].[CompanyName] AS [CompanyName],
[Extent1].[ContactName] AS [ContactName],
[Extent1].[ContactTitle] AS [ContactTitle],
[Extent1].[Address] AS [Address],
[Extent1].[City] AS [City],
[Extent1].[Region] AS [Region],
[Extent1].[PostalCode] AS [PostalCode],
[Extent1].[Country] AS [Country],
[Extent1].[Phone] AS [Phone],
[Extent1].[Fax] AS [Fax],
[Extent1].[RowVersion] AS [RowVersion]
FROM [dbo].[Customer] AS [Extent1]
WHERE [Extent1].[CompanyName] LIKE N'A%'
```

Note that there is no risk of a SQL injection attack with an EntityQuery. Query composition is not done using string manipulation, and SQL is generated by the ADO.NET provider from the LINQ query expression.

Some queries or query types will not generate parameterized SQL:

- dynamic queries
- paging queries

Canonical functions

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. Using DevForce, beginning in version 7.1.0, certain CLR methods are mapped to canonical functions and will be translated to the correct corresponding store function.

Many string manipulation methods have corresponding database functions. We saw that above with the *StartsWith* method, which is translated into the *Like* canonical function. Here's another example with *ToUpper*, which in SQL Server is translated into the *Upper* operator.

```
string country = "GERMANY";
var query = entityManager.Customers.Where(c => c.Country.ToUpper() == country);
var result = query.ToList();
```

```
exec sp_executesql N'SELECT
[Extent1].[Id] AS [Id],
[Extent1].[CompanyName] AS [CompanyName],
[Extent1].[ContactName] AS [ContactName],
[Extent1].[ContactTitle] AS [ContactTitle],
[Extent1].[Address] AS [Address],
[Extent1].[City] AS [City],
[Extent1].[Region] AS [Region],
[Extent1].[PostalCode] AS [PostalCode],
[Extent1].[Country] AS [Country],
[Extent1].[Phone] AS [Phone],
[Extent1].[Fax] AS [Fax],
[Extent1].[RowVersion] AS [RowVersion],
[Extent1].[CrtnTs] AS [CrtnTs],
[Extent1].[CrtnUsrId] AS [CrtnUsrId],
[Extent1].[ModTs] AS [ModTs],
[Extent1].[ModUsrId] AS [ModUsrId]
FROM [dbo].[Customer] AS [Extent1]
WHERE (UPPER([Extent1].[Country])) = @p__linq__0',N'@p__linq__0 nvarchar(4000)',@p__linq__0=N'GERMANY'
```

It's not necessary to use parameters with canonical functions, but you'll probably get better performance if you do.

Here's another example, this time using a *DateTime* canonical function.

```
var query = entityManager.OrderSummaries.Where(o=> o.OrderDate.Value.Year == DateTime.Now.Year);
var result = query.ToList();
```

The generated SQL (in SQL Server):

```
SELECT
[Extent1].[Id] AS [Id],
[Extent1].[CustomerId] AS [CustomerId],
[Extent1].[EmployeeId] AS [EmployeeId],
[Extent1].[OrderDate] AS [OrderDate],
[Extent1].[RequiredDate] AS [RequiredDate],
[Extent1].[ShippedDate] AS [ShippedDate],
[Extent1].[ShipperId] AS [ShipperId],
[Extent1].[Freight] AS [Freight],
[Extent1].[ShipName] AS [ShipName],
[Extent1].[ShipAddress] AS [ShipAddress],
[Extent1].[ShipCity] AS [ShipCity],
[Extent1].[ShipRegion] AS [ShipRegion],
[Extent1].[ShipPostalCode] AS [ShipPostalCode],
[Extent1].[ShipCountry] AS [ShipCountry],
[Extent1].[RowVersion] AS [RowVersion],
[Extent1].[CrtnTs] AS [CrtnTs],
[Extent1].[CrtnUsrId] AS [CrtnUsrId],
[Extent1].[ModTs] AS [ModTs],
[Extent1].[ModUsrId] AS [ModUsrId]
```

```
FROM [dbo].[OrderSummary] AS [Extent1]
WHERE (DATEPART (year, [Extent1].[OrderDate])) = (DATEPART (year, SysDateTime()))
```

VB developers: You'll need to add the *DateInterval* as a **known type** if using it in an n-tier application.