Query Execution

03/30/2017 • 9 minutes to read • +5

In this article

Deferred query execution

Immediate Query Execution

Store Execution

Store Exceptions

Store Configuration

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":

C#　　Copy

```
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);
    }
}
```

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.

C#

Copy

```
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);
    }
}
```

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.

C# Copy

```csharp
int orderID = 51987;

IQueryable<SalesOrderHeader> 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.

C# Copy

```csharp
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);
        }
    }
}
```

Casting Literals on the Client

Casting from `null` to a CLR type is executed on the client:

C#

Copy

```
IQueryable<Contact> query =
    from c in context.Contacts
    where c.EmailAddress == (string)null
    select c;
```

Casting to a type, such as a nullable Decimal, is executed on the client:

C#

Copy

```
var weight = (decimal?)23.77;
IQueryable<Product> 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:

C#

Copy

```
var weight = new decimal(23.77);
IQueryable<Product> 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 0.3333333333333333333333333333 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:

C# Copy

```
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
"));
}
```