# C# Support Session

## Introduction

C# Language has been evolving really fast in the last decade, the introduction of events and delegates, generics, anonymous delegates and lambdas, iterators, expression trees, etc.. Now any similarity between the code written in C# and Java is long gone.

The C# team has proven their talent in embracing other programming paradigms (Object Oriented, Component Oriented, Functional, Dynamic…) while keeping the language consistent and elegant. Now C# code is more expressive, less error-prone and much more fun to write.

Signum Framework makes intense uses of these new features, so in order to use the framework a high knowledge of C# is necessary.

In this tutorial we will try to put light in some dark parts of the new features in C#, for a complete explanation, I recommend the book:

C# in Depth, Second Edition  
  
Jon Skeet  
*Foreword by Eric Lippert*  
  
November, 2010 | 584 pages   
ISBN: 9781935182474

## Memory Model

When dealing with multithreading applications (like a web or windows server) is important to understand what data is being shared.

Do threads share memory? What memory? In order to answer these questions is useful to have a mental model of the two main important regions of the Memory:

**Stack:** Theplace where the parameters and local variables of your methods get stacked. Every function call has a stack frame and in the case of recursion the same function can have many. The size of an stack frame is known at compile-time, and when you return from a call the stack frame memory is released.

**Heap:** A pool where all the reference objects live and are accessed through references. You can reference objects of different sizes; this enables strings, arrays and polymorphism. The Garbage Collector releases the objects that are not referenced.

**Static Variables:** A ‘static’ amount of global variables that will never be released.

With this concepts clear we can defined that a Reference Type is a type that leaves in the Heap and will be accessed through a reference. I.E: classes, string, arrays…

A Value Type, on the other side, is stored whenever is declared, inside of the stack if it’s a local variable, as a static variable, or as a field inside of an object in the heap. I.E: int, DateTime, enum…

In the CLR, theoretically, everything inherits from System.Object that is a reference type, including value types like int that, by definition, have no inheritance support.

When we try to assign a value type (like int) to a reference type variable (like object) boxing is produced. A new object in the heap is created with the content of our value type, and then our variable points to this object.

object obj = 3; // BOXING HERE!

int val = (int)obj;//UNBOXING HERE!

### Sharing and isolating objects between threads

A thread is just a new stack. The new thread will have isolated local variables and parameters, but we share the Heap, and the Static Variables.

Thus, since nowadays most of the state is in reference objects, we have to be careful with:

* Objects accessed through static variables (like the Application object in ASP.Net)
* Objects passed as a parameter to threads
* Objects captured in the thread anonymous delegate (PLinq)

In this case we will need some synchronization primitive like Monitors.

Another cool feature is the ThreadStaticAttribute, when you place [ThreadStatic] over a static field, the field value from one thread becomes independent from the other thread, even if it looks like it’s a global variable.

Thread static fields are really useful for storing values like CurrentUser, CurrentConnection, CurrentTransaction… Let’s update our diagram with the new information.

## Delegates & Events

Delegates and events are resources in C# to solve dependency problems in our code, just like interfaces.

Most of the developers get the work done using events and delegates: many know how to subscribe to an event and some of them how to create their own, but very few have a clear understanding of what an event and a delegate is:

Delegates

**A delegate is a Type:** Just as string, int or Button. We have to declare it and we can make variables of it.

public delegate void Action();

(…)

public void Delegate()

{

Console.WriteLine(typeof(Action).Name);

Action a = null;

}

**A delegate is a strongly-typed pointer to a function:** In opposition to C++ function pointers, C# delegates are strongly typed so you can only create a delegate object with a function that fits exactly in the signature, and you can only call it with the right parameters:

Action a = new Action(Console.Beep); //Compiles ok

Action a = new Action(Console.Write); //COMPILE ERROR: No overload for 'Write' matches delegate 'System.Action'

a(); //Invokes OK

a("Hi!"); //COMPILE ERROR: Delegate 'Action' does not take 1 argument

C# 2.0 can also convert implicitly a ‘method group’ to a delegate that points to the right overload, so you can just write:

Action a = Console.Beep; //explicit delegate

**In C#, all delegates inherit from MultiCastDelegate:** So all delegates are actually a list of delegates and can potentially call more than one method when invoked (sequentially):

Action a = Console.Beep;

a += Console.Clear;

Action b = Console.WriteLine;

Action c = a + b;

c();//Calls Beep, Clear and WriteLine, in that order

Even on delegates that return values, all of them get invoked and only the last value is returned.

Func<string> f = null;

f += Console.ReadLine;

f += Console.ReadLine;

f += Console.ReadLine;

Console.WriteLine(f()); //Asks the user for 3 sentences, writes last

If you would like to write all the values for example, you would do it like this:

foreach (Func<string> fi in f.GetInvocationList())

{

Console.WriteLine(fi());

}

Events

**An event is a member:** Just as a field, method or property. We have to declare it inside of a class, and can be instance of static.

public class Button

{

public event EventHandler Click;

}

**An event is a special property for delegates:** It allows you to add or remove handlers, but not to assign to it or execute it:

public class Button

{

public event EventHandler Click;

public void DoClick()

{

Click(null, null); //Compiles

}

}

static void EventSample()

{

Button b = new Button();

b.Click(null, null); //COMPILE ERROR: The event 'Button.Click' can only appear on the left hand side of += or -= (except when used from within the type 'Button')

}

**An implicit event creates a hidden delegate:** In fact if we use .Net Reflector we will see code that looks like this (slightly modified for simplicity):

public class Button

{

// Fields

private EventHandler click;

// Events

public event EventHandler Click

{

add

{

this.click += value;

}

remove

{

this.click -= value;

}

}

// Methods

public void DoClick()

{

this.click(null, null);

}

}

As we see, when you call the event inside of the class, you’re actually calling the delegate instead. An implicit event is very similar to an auto-implemented property:

public string Name { get; set; }

string name;

public string Name

{

get { return this.name; }

set { this.name = value; }

}

public event EventHandler Click

EventHandler click;

public event EventHandler Click

{

add { this.click += value; }

remove { this.click -= value; }

}

Personally, I think a more consistent syntax will make it easier to grasp the concept of events for C# developers:

public ~~event~~ EventHandler Click { add; remove;}

Now that we know that, we can make events that store the delegates somewhere else, for example in a dictionary to save space for rarely used events.

### Event vs Delegate fields

Not that we have a clear understanding of events and delegates, and we know that we can use delegates by itself, let’s make clear the difference between am event and a delegate field.

static void EventSample()

{

Button b = new Button();

b.Click += ClickMethod;

b.Click -= ClickMethod;

b.Click = null; //COMPILE ERROR: The event 'Button.Click' can only appear on the left hand side of += or -=

b.Click(null, null); //COMPILE ERROR: The event 'Button.Click' can only appear on the left hand side of += or -=

b.Click2 += ClickMethod;

b.Click2 -= ClickMethod;

b.Click2 = null;

b.Click2(null, null);

}

Using an event we only let the consumers of the class to subscribe/unsubscribe, but if use a delegate field, they will also be able to:

* Assign null, effectively unsubscribing everybody.
* Replace de delegate (list) with a new one.
* Invoke de delegate from the outside.

For the classical use of delegates (Controls in the user interface) this features are dangerous, since events don’t return a value and controls don’t depend on the execution of the events, they just notify.

In Signum Framework, however, we sometimes use public delegate fields as a poor-man inversion of control system system, and sometimes delegates return values and doesn’t make sense to have more than one subscription, in this case being able to assign, instead of subscribe is interesting, so we just use public delegate fields.

## Generics

Generics allow writing code in a generic way, and then *concretize* it before using it. It has some benefits:

* **Self-Documented code:** If a code returns an ArrayList, you have to comment that there will be oranges inside, if it returns List<Orange> you don’t need to.
* **Less error-prone:** You save run-time errors because the CLR guarantees that no Apple will get into the List<Orange> where in an ArrayList is up to you writing solid code.
* **More efficient:** Especially with value types, where you save the boxing winning in memory locality, otherwise you also save the casting.

In C# there are three types that can be generic:

* **Class:** A class can be declared as generic, and then the parameter type is available within the class. You don’t need to decorate every method with the generic parameter.

public class Bag<T> // Generic class

{

T[] elements;

void Clear() //Method in a generic class

{

elements = null;

}

}

When using the generic type, you just have to give the generic argument a type.

new Bag<int>().Clear();

The static variables defined in generic types are **independent** for each instantiated type; this makes perfect sense since the variable type could be the generic parameter. Because of this, sometimes it makes sense to have a generic static class, for example to make a global list of the best objects of each type:

public static class Best<T>

{

public static T[] Elements;

}

Best<int>.Elements = new int[] { 1, 2, 42 };

Best<string>.Elements = new string[] { "hola", "hello", "hallo" };

* **Structs:** Structures (value types) can also be generics, being Nullable<T> the best known example:

public struct Nullable<T> where T : struct

* **Interfaces:**

public struct IComparable<T>

* **Delegates:** Even something like delegates can be generic, with some well-known examples like

public delegate void Action<in T>(T obj)

public delegate TResult Func<out TResult>()

There’s also just one kind of member that can be generic:

* **Methods: Both non-generic as generic classes (or structs) can also have methods that** include new generic parameters.

public void Switch<T>(ref T a, ref T b)

{

T temp = a;

a = b;

b = temp;

}

The consumer of the method can explicitly write the type of the method, or let the compiler infer it.

int a = 2, b = 3;

Switch<int>(ref a, ref b); //Implicit

Switch(ref a, ref b); //Explicit

In the case of methods with more than one generic parameter, the compiler is able to infer them all, but there’s no syntax to partially infer it.

static R Convert<T, R>(T value)

{

return (R)System.Convert.ChangeType(value, typeof(R));

}

Convert<int, long>(2); //Works

Convert(2); //COMPILE ERROR: The type arguments for method 'Convert<T,R>(T)' cannot be inferred from the usage. Try specifying the type arguments explicitly.

Convert<?, long>(2); //No synthax available

## Anonymous methods and lambdas

Anonymous methods and lambdas are two different syntaxes with pretty much the same functionality: writing functions in-line that are able to capture local context.

Let’s see the different syntaxes available:

Func<int, int, int> f;

//Anonymous method

f = delegate(int a, int b) { return a + b; }; //with parameters

f = delegate { return 3; }; //omitted parameters

//statement lambda

f = (int a, int b) => { return a + b; }; //explicit parameter types

f = (a, b) => { return a + b; }; //inferred parameter types

//expression lambda

f = (int a, int b) => a + b; //explicit parameter types

f = (a, b) => a + b; //inferred parameter types

### Type inference for lambdas

There have always been some kind of type inference, in almost any language every sub-expression has a compile-time type, but we don’t have to write it explicitly. For example the expression:

string str = "Hi! " + 5;

Doesn’t need any casting because the compiler knows that adding a string and an int results in another string, but until C# 2.0 and 3.0 this ‘deductions’ were constrained to the expression sub-tree: Type type of the expression depends on what it’s made of, not on where it’s going to be assigned to.

Now, these rules have been relaxed to make lambda syntax more succinct, this is how it works more or less:

f = (a, b) => a + b;

//Since Func<int, int, int> signature is int(int, int) we can deduce the type parameters

f = (int a, int b) => a + b;

//Since int + int returns int, the inference worked, otherwise try other options (overloads)

//let's also makes the delegate explicit

f = new Func<int, int, int>((int a, int b) => a + b);

Note how, in order to know the type of “(a, b) => a + b;” we need to look at the type of “f”, outside of the expression itself.

### Type inference for local variables and anonymous classes

C# also has the ability to infer type for local variables that have a value as part of the declaration, using ‘var’ as the type instead:

var b = "hi!";

Console.WriteLine(b.Substring(2));

This feature convenient for expressions that return long generic types:

var dictionary = new Dictionary<Country, Dictionary<Town, ZipCode>>();

But is a must when dealing with anonymous classes: classes defined on-the-fly just to hold some values, usually as a result of a LINQ query:

var c = new

{

Name = "John",

Length = 4

};

Console.WriteLine(c.Name);

Even if the C# compiler knows about the type, and can provide IntelliSense for it, there’s no name for the type so a variable couldn’t be declared without this feature.

### Type inference tricks

Let’s puss not type inference a little bit forward. Sometimes you do something like this:

var men = "John,Richard,Luis".Split(',').Select(a => new

{

Name = a,

Length = a.Length

});

var women = "Eva,Anna,Stephanie".Split(',').Select(a => new

{

Name = a,

Length = a.Length

});

But you see the redundancy between the two list and you would like to factor-out the select projector, that is identical:

Func<string, ?> projector = a => new

{

Name = a,

Length = a.Length

};

var men = "John,Richard,Luis".Split(',').Select(projector);

var women = "Eva,Anna,Stephanie".Split(',').Select(projector);

Unfortunately, there’s no syntax for partially infer the return type of projector. Even if we explicitly declare the parameter:

//COMPILE ERROR: Cannot assign lambda expression to an implicitly-typed local variable

var projector = (string a) => new

{

Name = a,

Length = a.Length

};

Now, even if the C# compiler can compile the lambda itself, it cannot convert it to the appropriate delegate, since Func<> and Action<> delegates are not especially hard-coded in the compiler in any way.

What we need is to tell the compiler that we want a Func<,> and let him does the type inference. We can do that using a helper dummy function:

public static Func<R, T> FuncHelper<R,T>(Func<R,T> f)

{

return f;

}

As we see, the method does nothing at all, but gives the compiler the necessary information so now we can do:

var projector = FuncHelper((string a) => new

{

Name = a,

Length = a.Length

});

var men = "John,Richard,Luis".Split(',').Select(projector);

var women = "Eva,Anna,Stephanie".Split(',').Select(projector);

And now everything compiles ok and we were able to factor-out our anonymous class projector!

## Expression trees

When making Linq against any remote source, typically a database, we don’t want to enumerate all the items and filter them out locally, instead we want to give the remote source enough information so it can make the operations and give us the result (less transfer).

In order to do that, keeping a consistent syntax, the C# compiler need to be able to generate, instead of an executable method that can be run locally, an expression that can be sent to the server that expresses the programmer intentions.

Currently, there’s just one case when the C# compiler generates this expression trees, when an **expression lambda (not statement lambdas jet)** is going to be assigned into an Expression<T> variable.

Func<int, int> f = a => a \* 2;

f(4); //Method that can be executed

Expression<Func<int, int>> e = a => a \* 2;

Console.WriteLine(e.ToString()); //writes "a => a \* 2"

Console.WriteLine(e.Body.ToString()); //wites "a \* 2"

e.Compile()(4); //Can be compiler and then executed

As you can see, the exact same lambda expression can be compiled to a method, or an expression tree depending on what the target. Once is an expression tree we can:

* Call ToString to see a *C-Sharpish* representation of the expression
* Navigate through the expression objects (ConstantExpression, ParameterExpressoin, BinaryExpression…)
* Compile the lambda and execute it!

### Uses of Expression Trees

**Strongly typed reflection:** Pointing to a member using reflection could be error prone since you rely on nasty string literals. Instead you can prepare your API to take Expression Trees and look for the Member in the generated expression. This way the user has IntelliSense, Refactoring and compile-type errors when interacting with your library. Signum Framework uses this technique intensively.

**Meta-Programming:** Expression trees are not restricted to be generated by the compiler. They can also be run-time generated and then compiled, effectively making an efficient meta-programming feature. This is especially interesting to cache and re-use pieces of code generated by reflection. This way, you pay for the performance penalty of reflection once, and then is as performant as if it would have been written by hand.

### Immutability

What we cannot do with expression trees, however, is modifying them. They are immutable and once constructed cannot change; instead we have to create a new one. For example in order to change this expression:

Expression<Func<int, int>> e = a => a \* 2;

So that is multiplied by 3 instead of 2, we cannot do this:

((BinaryExpression)e.Body).Right = Expression.Constant(3);

//COMPILE ERROR: Property or indexer 'Right' cannot be assigned to -- it is read only

Instead we have to build all the affected nodes again:

e = Expression.Lambda<Func<int, int>>(Expression.Multiply(e.Parameters[0], Expression.Constant(3)), e.Parameters);

This property of expression trees makes it easier to reason about expressions when there are complex transformations (like in a LINQ provider), but makes it really inconvenient without the use of Visitors (next chapter).

### Expression Trees and Queryable

If IEnumerable<T> is the type that enables LINQ on local objects, IQueryable<T> is the equivalent for remote ones. When you write a query against an IQueryable, you use the extension methods defined in Queryable, that take an Expression<> instead of a delegate.

That’s why the syntax is so similar, and also is so easy to forget that you are talking with a database.

One interesting detail, while the lambda expressions passed as an arguments to the Queryable method are generated at compile-time, the expression that defines the whole query is generated at runtime as you call the LINQ operators (Where, Select…).

## Exercises

**Actions**

Action<int> write = Console.WriteLine;

Action a1 = Console.Beep;

a1();

Action a2 = () => Console.Beep();

a2();

Action<int> a3 = a => Console.WriteLine(a);

a3(2);

Action<Func<int>> a4 = f => Console.WriteLine(DateTime.Today.Day == 1 ? 1 : f());

a4(() => 4);

Action que reciba un function (DateTime->int)

**Funcs**

Func<int> f1= ()=>3;

write(f1());

Func<int, int> f2 = a => a + 2;

write(f2(3));

Func<int, int, int> f3 = (a,b) => a + b;

write(f3(1, 2));

Func<int, Func<int, int>> f4 = a => b => a + b;

write(f4(1)(2));

Func<Func<int, int>, int> f5 = fun => fun(3);

write(f5(a => a \* 2));

**Expressions**

Expression<Func<int>> e1 = () => 3;

write(e1.Compile()());

Expression<Func<int, int>> e2 = a => a + 2;

write(e2.Compile()(2));

Expression<Func<int, int, int>> e3 = (a, b) => a + b;

write(e3.Compile()(2, 3));

Expression<Func<int, Func<int, int>>> e4 = a => b => a + b;

write(e4.Compile()(2)(3));

Expression<Func<Func<int, int>, int>> e5 = fun => fun(3);

write(e5.Compile()(a => a \* 2));

Expression<Func<int, Expression<Func<int, int>>>> e6 = a => b => a + b;

write(e6.Compile()(2).Compile()(3));

Expression<Func<Expression<Func<int, int>>, int>> e7 = exp => exp.Compile()(3);

write(e7.Compile()(a => a \* 2));

**Capturing variables (not values!)**

int value = 3;

Func<int> f = () => value;

value = 4;

Console.WriteLine(f()); //Writes 4!

Action action = () => value++;

action();

Console.WriteLine(value); //Writes 5;

**Recursion**

static void Recursion()

{

Func<int, long> fib = null;

fib = a => a <= 1 ? a : (fib(a - 1) + fib(a - 2));

Console.WriteLine(fib(10)); //Writes 55

Console.WriteLine(fib(100)); //Takes forever!

}

**Caching results**

public static Func<T, R> Cache<T, R>(Func<T, R> func)

{

Dictionary<T, R> cache = new Dictionary<T, R>();

return a =>

{

R result;

if (!cache.TryGetValue(a, out result))

{

result = func(a);

cache[a] = result;

}

return result;

};

}

static void Recursion()

{

Func<int, long> fib = null;

fib = a => a <= 1 ? a : (fib(a - 1) + fib(a - 2));

Console.WriteLine(fib(10)); //Writes 55

fib = Cache(fib);

Console.WriteLine(fib(100)); //Writes 3.736.710.778.780.434.371 really fast!

}

# LINQ Provider Implementation

## Black box model

If you take Signum Linq provider (or any other) as a black box, it’s a library where you write a LINQ query and it returns the results.

Since no database has ‘native’ LINQ support, translating the query to SQL is mandatory.

Once the query is executed, our ADO.Net will return a DataTable, or a DataReader, but not the Anonymous Type you want as a result. It’s necessary then that the LINQ provider also makes this conversion.

Finally, since usually we will use the same query with different parameters, we need to parameterize the SQL to please the DBMS execution plan cache, and to protect our code against SQL Injection.

So all in all, here’s our black box, with this three things:

Note: Sometimes, for more complex queries where each item in the result has collections, a more complex thing happens: The Translate Query black box returns more than one of this ‘trios’, they get executed independently and then the results get merged.

From this simplistic point of view, basically we have to split the query in three things:

* What to evaluate before the query gets executed.
* What to should the DBMS execute.
* What to do with the results once is executed.

Note that in a LINQ query you don’t explicitly determine these boundaries, and there’s a certain level of indetermination that is solved with heuristics. I.E:

Query<Person>()

.Where(a => a.Name == "Jo" + "hn")

.Select(a => a.Surname + DateTime.Now).ToList();

In this query, should we send two parameters (“Jo”, “hn”) and make the concatenation in SQL, or should we send just one (“John”).

And on the results, should we use C# DateTime.Now?, or SQL GetDate() one?

## Visitors

As we seen before, modifying expression is a painful experience since you have to replace all the affected nodes in the expression (from a leaf, all the way up to the root). And since a LINQ provider is all about modifying expressions, some tool is necessary.

This tool is called ExpressionVisitors. This visitor offers the functionality of walking all the expression tree recursively and, if some element changes (by returning a different expression), automatically propagate the change up. The visitor is structured as a battery of overrideable methods, one for each type of expression node.

Here is an example of a visitor for a super-simplified version of expression trees:

public abstract class ExpressionVisitor

{

protected virtual Expression Visit(Expression exp)

{

if (exp == null)

return exp;

switch (exp.NodeType)

{

case ExpressionType.Constant:

return this.VisitConstant((ConstantExpression)exp);

case ExpressionType.Parameter:

return this.VisitParameter((ParameterExpression)exp);

case ExpressionType.Add:

case ExpressionType.Subtract:

case ExpressionType.Multiply:

case ExpressionType.Divide:

return this.VisitBinary((BinaryExpression)exp);

default:

throw new Exception(exp.NodeType.ToString());

}

}

protected virtual Expression VisitBinary(BinaryExpression b)

{

Expression left = this.Visit(b.Left);

Expression right = this.Visit(b.Right);

if (left != b.Left || right != b.Right)

{

return Expression.MakeBinary(b.NodeType, left, right);

}

return b;

}

protected virtual Expression VisitConstant(ConstantExpression c)

{

return c;

}

protected virtual Expression VisitParameter(ParameterExpression p)

{

return p;

}

}

Finally, in order to make a visitor that replaces every constant by the value ‘3’ we just need to write:

public class ReplaceConstantsVisitor: ExpressionVisitor

{

public static Expression Replace(Expression exp)

{

return new ReplaceConstantsVisitor().Visit(exp);

}

protected override Expression VisitConstant(ConstantExpression c)

{

if (c.Type == typeof(int))

return Expression.Constant(3);

return base.VisitConstant(c);

}

}

And then we can call it like this:

Expression<Func<int,int>> f = p => ((2 + p) \* 5) - 1;

Expression replacedBody = ReplaceConstantsVisitor.Replace(f.Body);

Console.WriteLine(replacedBody.ToString());//(((3 + p) \* 3) - 3)

## Translation Pipeline

Jut as a matter of curiosity, these are the stages of an expression across Linq to Signum:

### C# Stage

In this stage the expression stills looks like a C# expression tree

**ExpressionCleaner:** The first visitor makes three related things:

* **Evaluate** independent sub-expression that can be passed as a parameter.
* **Simplify** unreachablecode due to boolean expression sort-circuitand conditional and coalesce operators.
* **Expand** expressions using the three expansion strategies in Linq to Signum (Expression Method/Properties, Invoke and MethodExpander)

**OverloadSimplifier:** Normalize overloads of methods to simplify next stages.

**QueryFilterer:** Filters out tables using EntityEvents<T>.FilterQuery event

### C# to SQL

**QueryBinder:** Transform the C# expression in a SQLish expression. Starts by translating each expression porinting to a table (Database.Query<T>) ProjetionExpression that contains:

* SelectExpression: Represents the SQL that will be executed
* Projector: Represents the lambda that will transform the columns in the resulting objects

Then, for each operator, binds the lambdas inside (selector, projector…) to database columns and creates a new Projector with an update SelectExpression and Projector.

Some lambdas (like select projector) can be partially evaluated in C#, this is called Partial Nomination.

Others lambdas (like where predicate) have to be completely translated to SQL, this is called FullNomination.

During the process of nomination, C# functions are sometimes translated to their SQL counterparts and smart comparisons between entities are translated.

At the end of this stage we have something that looks like a SQL evaluable query and a projector, but the query really complicated, with lots of columns and nested queries. The rest of the process consist in simplify this query to help the database.

### SQL Optimization

**AggregateRewriter:** Transforms general aggregations (MAX, COUNT…) in simple SQL aggregations if possible.

**QueryRebinder:** Ensuresthatall the columns in a nested query are exposed if they are necessary for some parent query or the projector.

**EntityCleaner:** Simplifies the ‘Entity Expressions’ releasing all the non-necessary columns (keeping only ID and ToStr).

**AliasProjectionReplacer:** Ensures that re-used projectors in the same query have independent aliases.

**CountOrderByRemover:** Removes OrderBy on queries that just want to return a count (special case, useful for pagination).

**UnusedColumnRemover:** Recursively removes all the unnecessary columns.

**RowNumberFiller:** Addsall the ordering columns to RowNumber expressions (useful for rankings and pagination).

**RedundantSubqueryRemover:** Collapses nested queries when possible.

**ChildProjectionFlattener:** Transforms nested projections that have references to the main projector in independent projections with the main projection embedded.

### SQL Optimization

**TranslatorBuilder:** For each projection (usually one) creates:

* **GetParameterExpressions:** Expression that get’s the SqlParameters from the ConstantExpressions initially evaluated from the context of the query (captured variables, static fields…).
* **CommandText:** ParametrizedSQL Query ready to get executed.
* **ProjectorExpression:** Expression that transforms the value returned by the SqlDataReader into objects.

Finally the query is executed and, in some cases, the Retriever is invoked to re-create the graph of entities (if there are full entities returned).