```
CREATE TABLE Purchase (
  ID int NOT NULL IDENTITY PRIMARY KEY,
  CustomerID int NOT NULL REFERENCES Customer(ID),
  Date datetime NOT NULL,
  Description nvarchar(30) NOT NULL,
  Price decimal NOT NULL
)
```

Overview

In this section, we provide an overview of the standard query operators. They fall into three categories:

- Sequence in, sequence out (sequence→sequence)
- Sequence in, single element or scalar value out
- Nothing in, sequence out (*generation* methods)

We first present each of the three categories and the query operators they include, and then we take up each individual query operator in detail.

Sequence→Sequence

Most query operators fall into this category—accepting one or more sequences as input and emitting a single output sequence. Figure 9-1 illustrates those operators that restructure the shape of the sequences.

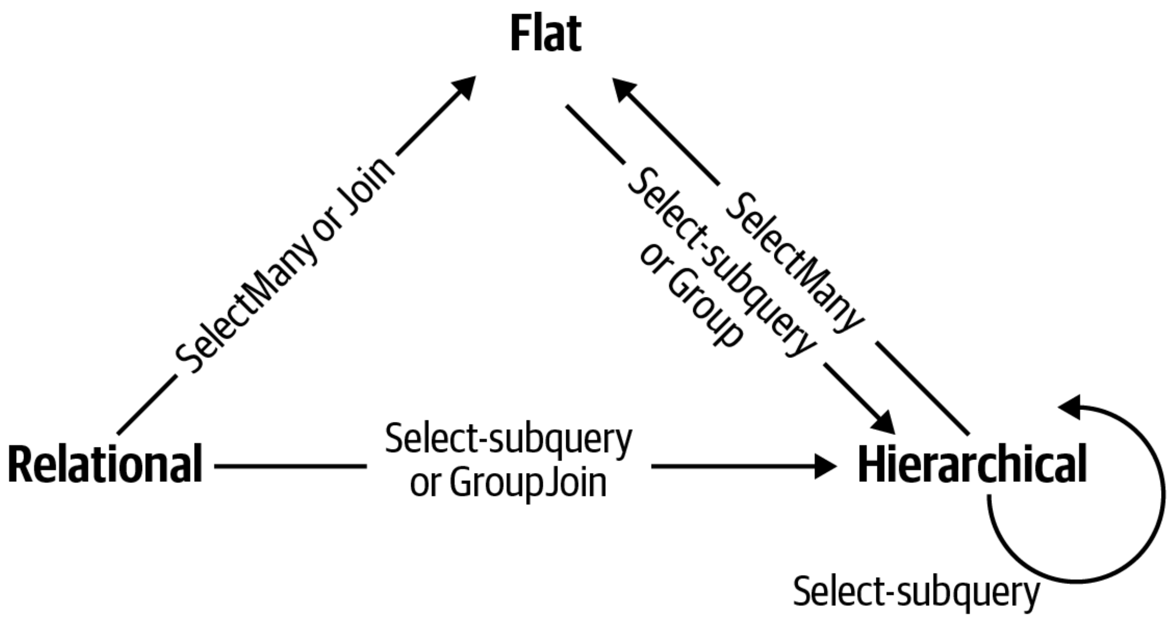

Figure 9-1. Shape-changing operators

Filtering

`IEnumerable<TSource>` →`IEnumerable<TSource>`

Returns a subset of the original elements.

```
Where, Take, TakeLast, TakeWhile, Skip, SkipLast, SkipWhile,
Distinct, DistinctBy
```

Projecting

`IEnumerable<TSource>`→`IEnumerable<TResult>`

Transforms each element with a lambda function. `SelectMany` flattens nested sequences; `Select` and `SelectMany` perform inner joins, left outer joins, cross joins, and non-equi joins with EF Core.

```
Select, SelectMany
```

Joining

`IEnumerable<TOuter>`, `IEnumerable<TInner>`→`IEnumerable<TResult>`

Meshes elements of one sequence with another. `Join` and `GroupJoin` operators are designed to be efficient with local queries and support inner and left outer joins. The `Zip` operator enumerates two sequences in step, applying a function over each element pair. Rather than naming the type arguments `TOuter` and `TInner`, the `Zip` operator names them `TFirst` and `TSecond`:

```
IEnumerable<TFirst>,
IEnumerable<TSecond>→IEnumerable<TResult>
```

```
Join, GroupJoin, Zip
```

Ordering

```
IEnumerable<TSource>→IOrderedEnumerable<TSource>
```

Returns a reordering of a sequence.

```
OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse
```

Grouping

```
IEnumerable<TSource>→IEnumerable<IGrouping<TKey,TElement>>
```

```
IEnumerable<TSource>→IEnumerable<TElement[]>
```

Groups a sequence into subsequences.

```
GroupBy, Chunk
```

Set operators

```
IEnumerable<TSource>,
IEnumerable<TSource>→IEnumerable<TSource>
```

Takes two same-typed sequences and returns their commonality, sum, or difference.

```
Concat, Union, UnionBy, Intersect, IntersectBy, Except, ExceptBy
```

Conversion methods: Import

`IEnumerable→IEnumerable<TResult>`

```
OfType, Cast
```

Conversion methods: Export

`IEnumerable<TSource>`→An array, list, dictionary, lookup, or sequence

```
ToArray, ToList, ToDictionary, ToLookup, AsEnumerable, AsQueryable
```

Sequence→Element or Value

The following query operators accept an input sequence and emit a single element or value.

Element operators

`IEnumerable<TSource>→TSource`

Picks a single element from a sequence.

```
First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault,
ElementAt, ElementAtOrDefault, MinBy, MaxBy, DefaultIfEmpty
```

Aggregation methods

`IEnumerable<TSource>→`*`scalar`*

Performs a computation across a sequence, returning a scalar value (typically a number).

```
Aggregate, Average, Count, LongCount, Sum, Max, Min
```

Quantifiers

IEnumerable<TSource>→*bool*

An aggregation returning true or false.

```
All, Any, Contains, SequenceEqual
```

Void→Sequence

In the third and final category are query operators that produce an output sequence from scratch.

Generation methods

void→IEnumerable<TResult>

Manufactures a simple sequence.

```
Empty, Range, Repeat
```

Filtering

IEnumerable<TSource>→IEnumerable<TSource>

Method	Description	SQL equivalents
`Where`	Returns a subset of elements that satisfy a given condition	`WHERE`
`Take`	Returns the first `count` elements and discards the rest	`WHERE ROW_NUMBER()...` *or* `TOP` *`n`* subquery
`Skip`	Ignores the first `count` elements and returns the rest	`WHERE ROW_NUMBER()...` *or* `NOT IN (SELECT TOP` *`n`*`...)`
`TakeLast`	Takes only the last `count` elements	Exception thrown
`SkipLast`	Ignores the last `count` element	Exception thrown
`TakeWhile`	Emits elements from the input sequence until the predicate is false	Exception thrown
`SkipWhile`	Ignores elements from the input sequence until the predicate is false, and then emits the rest	Exception thrown
`Distinct`, `DistinctBy`	Returns a sequence that excludes duplicates	`SELECT DISTINCT...`

> **NOTE**
>
> The "SQL equivalents" column in the reference tables in this chapter do not necessarily correspond to what an `IQueryable` implementation such as EF Core will produce. Rather, it indicates what you'd typically use to do the same job if you were writing the SQL query yourself. Where there is no simple translation, the column is left blank. Where there is no translation at all, the column reads "Exception thrown."
>
> `Enumerable` implementation code, when shown, excludes checking for null arguments and indexing predicates.

With each of the filtering methods, you always end up with either the same number or fewer elements than you started with. You can never get more! The elements are also identical when they come out; they are not transformed in any way.

Where

Argument	Type
Source sequence	`IEnumerable<TSource>`
Predicate	`TSource => bool or (TSource,int) => bool`[a]

[a] Prohibited with LINQ to SQL and Entity Framework

Query syntax

```
where bool-expression
```

Enumerable.Where implementation

The internal implementation of `Enumerable.Where`, null checking aside, is functionally equivalent to the following:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func <TSource, bool> predicate)
{
  foreach (TSource element in source)
    if (predicate (element))
      yield return element;
}
```

Overview

`Where` returns the elements from the input sequence that satisfy the given predicate.

For instance:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> query = names.Where (name => name.EndsWith ("y"));

// Harry
// Mary
// Jay
```

In query syntax:

```
IEnumerable<string> query = from n in names
                            where n.EndsWith ("y")
                            select n;
```

A `where` clause can appear more than once in a query and be interspersed with `let`, `orderby`, and `join` clauses:

```
from n in names
where n.Length > 3
let u = n.ToUpper()
where u.EndsWith ("Y")
select u;

// HARRY
// MARY
```

Standard C# scoping rules apply to such queries. In other words, you cannot refer to a variable prior to declaring it with a range variable or a `let` clause.

Indexed filtering

`Where`'s predicate optionally accepts a second argument, of type `int`. This is fed with the position of each element within the input sequence, allowing the predicate to use this information in its filtering decision. For example, the following skips every second element:

```
IEnumerable<string> query = names.Where ((n, i) => i % 2 == 0);

// Tom
// Harry
// Jay
```

An exception is thrown if you use indexed filtering in EF Core.

SQL LIKE comparisons in EF Core

The following methods on `string` translate to SQL's `LIKE` operator:

```
Contains, StartsWith, EndsWith
```

For instance, `c.Name.Contains ("abc")` translates to `customer.Name LIKE '%abc%'` (or more accurately, a parameterized version of this). `Contains` lets you compare only against a locally evaluated expression; to compare against another column, you must use the `EF.Functions.Like` method:

```
... where EF.Functions.Like (c.Description, "%" + c.Name + "%")
```

`EF.Functions.Like` also lets you perform more complex comparisons (e.g., `LIKE 'abc%def%'`).

< and > string comparisons in EF Core

You can perform *order* comparison on strings with string's CompareTo method; this maps to SQL's < and > operators:

```
dbContext.Purchases.Where (p => p.Description.CompareTo ("C") < 0)
```

WHERE x IN (…, …, …) in EF Core

With EF Core, you can apply the Contains operator to a local collection within a filter predicate. For instance:

```
string[] chosenOnes = { "Tom", "Jay" };

from c in dbContext.Customers
where chosenOnes.Contains (c.Name)
...
```

This maps to SQL's IN operator. In other words:

```
WHERE customer.Name IN ("Tom", "Jay")
```

If the local collection is an array of entities or nonscalar types, EF Core might instead emit an EXISTS clause.

Take, TakeLast, Skip, SkipLast

Argument	Type
Source sequence	IEnumerable<TSource>
Number of elements to take or skip	int

`Take` emits the first *n* elements and discards the rest; `Skip` discards the first *n* elements and emits the rest. The two methods are useful together when implementing a web page allowing a user to navigate through a large set of matching records. For instance, suppose that a user searches a book database for the term “mercury”, and there are 100 matches. The following returns the first 20:

```
IQueryable<Book> query = dbContext.Books
  .Where   (b => b.Title.Contains ("mercury"))
  .OrderBy (b => b.Title)
  .Take (20);
```

The next query returns books 21 to 40:

```
IQueryable<Book> query = dbContext.Books
  .Where   (b => b.Title.Contains ("mercury"))
  .OrderBy (b => b.Title)
  .Skip (20).Take (20);
```

EF Core translates `Take` and `Skip` to the `ROW_NUMBER` function in SQL Server 2005, or a `TOP` *n* subquery in earlier versions of SQL Server.

The `TakeLast` and `SkipLast` methods take or skip the last *n* elements.

From .NET 6, the `Take` method is overloaded to accept a `Range` variable. This overload can subsume the functionality of all four methods; for instance, `Take(5..)` is equivalent to `Skip(5)`, and `Take(..^5)` is equivalent to `SkipLast(5)`.

TakeWhile and SkipWhile

Argument	Type
Source sequence	`IEnumerable<TSource>`
Predicate	`TSource => bool` or `(TSource,int) => bool`

`TakeWhile` enumerates the input sequence, emitting each item until the given predicate is false. It then ignores the remaining elements:

```
int[] numbers      = { 3, 5, 2, 234, 4, 1 };
var takeWhileSmall = numbers.TakeWhile (n => n < 100);   // { 3, 5, 2 }
```

`SkipWhile` enumerates the input sequence, ignoring each item until the given predicate is false. It then emits the remaining elements:

```
int[] numbers      = { 3, 5, 2, 234, 4, 1 };
var skipWhileSmall = numbers.SkipWhile (n => n < 100);   // { 234, 4, 1 }
```

`TakeWhile` and `SkipWhile` have no translation to SQL and throw an exception if used in an EF Core query.

Distinct and DistinctBy

`Distinct` returns the input sequence, stripped of duplicates. You can optionally pass in a custom equality comparer. The following returns distinct letters in a string:

```
char[] distinctLetters = "HelloWorld".Distinct().ToArray();
string s = new string (distinctLetters);                     // HeloWrd
```

We can call LINQ methods directly on a string because `string` implements `IEnumerable<char>`.

The `DistinctBy` method was introduced in .NET 6 and lets you specify a key selector to be applied before performing equality comparison. The result of the following expression is `{1,2,3}`:

```
new[] { 1.0, 1.1, 2.0, 2.1, 3.0, 3.1 }.DistinctBy (n => Math.Round (n, 0))
```

Projecting

`IEnumerable<TSource>`→ `IEnumerable<TResult>`

Method	Description	SQL equivalents
`Select`	Transforms each input element with the given lambda expression	`SELECT`
`SelectMany`	Transforms each input element, and then flattens and concatenates the resultant subsequences	`INNER JOIN`, `LEFT OUTER JOIN`, `CROSS JOIN`

NOTE

When querying a database, `Select` and `SelectMany` are the most versatile joining constructs; for local queries, `Join` and `GroupJoin` are the most *efficient* joining constructs.

Select

Argument	Type
Source sequence	`IEnumerable<TSource>`
Result selector	`TSource => TResult` or `(TSource,int) => TResult`[a]

[a] Prohibited with EF Core

Query syntax

```
select projection-expression
```

Enumerable implementation

```
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
{
  foreach (TSource element in source)
    yield return selector (element);
}
```

Overview

With `Select`, you always get the same number of elements that you started with. Each element, however, can be transformed in any manner by the lambda function.

The following selects the names of all fonts installed on the computer (from `System.Drawing`):

```
IEnumerable<string> query = from f in FontFamily.Families
                            select f.Name;

foreach (string name in query) Console.WriteLine (name);
```

In this example, the `select` clause converts a `FontFamily` object to its name. Here's the lambda equivalent:

```
IEnumerable<string> query = FontFamily.Families.Select (f => f.Name);
```

`Select` statements are often used to project into anonymous types:

```
var query =
  from f in FontFamily.Families
  select new { f.Name, LineSpacing = f.GetLineSpacing (FontStyle.Bold) };
```

A projection with no transformation is sometimes used with query syntax to satisfy the requirement that the query end in a `select` or `group` clause. The following selects fonts supporting strikeout:

```
IEnumerable<FontFamily> query =
  from f in FontFamily.Families
  where f.IsStyleAvailable (FontStyle.Strikeout)
  select f;

foreach (FontFamily ff in query) Console.WriteLine (ff.Name);
```

In such cases, the compiler omits the projection when translating to fluent syntax.

Indexed projection

The `selector` expression can optionally accept an integer argument, which acts as an indexer, providing the expression with the position of each input in the input sequence. This works only with local queries:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IEnumerable<string> query = names
  .Select ((s,i) => i + "=" + s);     //  { "0=Tom", "1=Dick", ... }
```

Select subqueries and object hierarchies

You can nest a subquery in a `select` clause to build an object hierarchy. The following example returns a collection describing each directory under `Path.GetTempPath()`, with a subcollection of files under each directory:

```
string tempPath = Path.GetTempPath();
DirectoryInfo[] dirs = new DirectoryInfo (tempPath).GetDirectories();

var query =
  from d in dirs
  where (d.Attributes & FileAttributes.System) == 0
  select new
  {
    DirectoryName = d.FullName,
    Created = d.CreationTime,

    Files = from f in d.GetFiles()
            where (f.Attributes & FileAttributes.Hidden) == 0
            select new { FileName = f.Name, f.Length, }
  };

foreach (var dirFiles in query)
{
  Console.WriteLine ("Directory: " + dirFiles.DirectoryName);
  foreach (var file in dirFiles.Files)
    Console.WriteLine ("  " + file.FileName + " Len: " + file.Length);
}
```

The inner portion of this query can be called a *correlated subquery*. A subquery is correlated if it references an object in the outer query—in this case, it references `d`, the directory being enumerated.

> **NOTE**
>
> A subquery inside a `Select` allows you to map one object hierarchy to another, or map a relational object model to a hierarchical object model.

With local queries, a subquery within a `Select` causes double-deferred execution. In our example, the files aren't filtered or projected until the inner `foreach` statement enumerates.

Subqueries and joins in EF Core

Subquery projections work well in EF Core, and you can use them to do the work of SQL-style joins. Here's how we retrieve each customer's name along with their high-value purchases:

```
var query =
  from c in dbContext.Customers
  select new {
               c.Name,
               Purchases = (from p in dbContext.Purchases
                            where p.CustomerID == c.ID && p.Price > 1000
                            select new { p.Description, p.Price })
                            .ToList()
             };

foreach (var namePurchases in query)
{
  Console.WriteLine ("Customer: " + namePurchases.Name);
  foreach (var purchaseDetail in namePurchases.Purchases)
    Console.WriteLine ("  - $$$: " + purchaseDetail.Price);
}
```

> **NOTE**
>
> Note the use of `ToList` in the subquery. EF Core 3 cannot create queryables from the subquery result when that subquery references the `DbContext`. This issue is being tracked by the EF Core team and might be resolved in a future release.

> **NOTE**
>
> This style of query is ideally suited to interpreted queries. The outer query and subquery are processed as a unit, preventing unnecessary round-tripping. With local queries, however, it's inefficient because every combination of outer and inner elements must be enumerated to get the few matching combinations. A better choice for local queries is `Join` or `GroupJoin`, described in the following sections.

This query matches up objects from two disparate collections, and it can be thought of as a "join." The difference between this and a conventional database join (or subquery) is that we're not flattening the output into a single two-dimensional result set. We're mapping the relational data to hierarchical data, rather than to flat data.

Here's the same query simplified by using the `Purchases` collection navigation property on the `Customer` entity:

```
from c in dbContext.Customers
select new
{
  c.Name,
  Purchases = from p in c.Purchases    // Purchases is List<Purchase>
              where p.Price > 1000
              select new { p.Description, p.Price }
};
```

(EF Core 3 does not require `ToList` when performing the subquery on a navigation property.)

Both queries are analogous to a left outer join in SQL in the sense that we get all customers in the outer enumeration, regardless of whether they have any purchases. To emulate an inner join—whereby customers without high-value purchases are excluded—we would need to add a filter condition on the purchases collection:

```
from c in dbContext.Customers
where c.Purchases.Any (p => p.Price > 1000)
```

```
select new {
             c.Name,
             Purchases = from p in c.Purchases
                         where p.Price > 1000
                         select new { p.Description, p.Price }
           };
```

This is slightly untidy, however, in that we've written the same predicate (`Price > 1000`) twice. We can avoid this duplication with a `let` clause:

```
from c in dbContext.Customers
let highValueP = from p in c.Purchases
                 where p.Price > 1000
                 select new { p.Description, p.Price }
where highValueP.Any()
select new { c.Name, Purchases = highValueP };
```

This style of query is flexible. By changing `Any` to `Count`, for instance, we can modify the query to retrieve only customers with at least two high-value purchases:

```
...
where highValueP.Count() >= 2
select new { c.Name, Purchases = highValueP };
```

Projecting into concrete types

In the examples so far, we've instantiated anonymous types in the output. It can also be useful to instantiate (ordinary) named classes, which you populate with object initializers. Such classes can include custom logic and can be passed between methods and assemblies without using type information.

A typical example is a custom business entity. A custom business entity is simply a class that you write with some properties but is designed to hide lower-level (database-related) details. You might exclude foreign key fields from business-entity classes, for instance. Assuming that we wrote custom entity classes called `CustomerEntity` and `PurchaseEntity`, here's how we could project into them:

```
IQueryable<CustomerEntity> query =
  from c in dbContext.Customers
  select new CustomerEntity
  {
    Name = c.Name,
    Purchases =
      (from p in c.Purchases
       where p.Price > 1000
       select new PurchaseEntity {
                                  Description = p.Description,
                                  Value = p.Price
                                }
      ).ToList()
  };

// Force query execution, converting output to a more convenient List:
List<CustomerEntity> result = query.ToList();
```

> **NOTE**
>
> When created to transfer data between tiers in a program or between separate systems, custom business entity classes are often called data transfer objects (DTO). DTOs contain no business logic.

Notice that so far, we've not had to use a `Join` or `SelectMany` statement. This is because we're maintaining the hierarchical shape of the data, as illustrated in Figure 9-2. With LINQ, you can often avoid the traditional SQL approach of flattening tables into a two-dimensional result set.

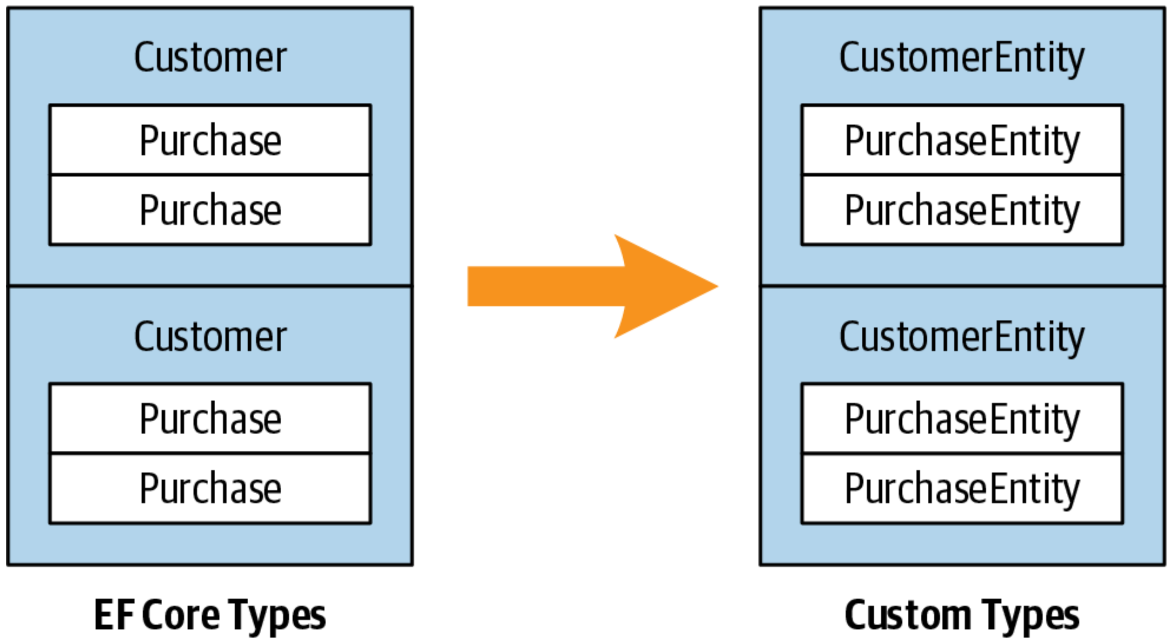

Figure 9-2. Projecting an object hierarchy

SelectMany

Argument	Type
Source sequence	`IEnumerable<TSource>`
Result selector	`TSource => IEnumerable<TResult>` or `(TSource,int) => IEnumerable<TResult>`[a]

[a] Prohibited with EF Core

Query syntax

```
from identifier1 in enumerable-expression1
from identifier2 in enumerable-expression2
...
```

Enumerable implementation

```
public static IEnumerable<TResult> SelectMany<TSource,TResult>
  (IEnumerable<TSource> source,
   Func <TSource,IEnumerable<TResult>> selector)
{
  foreach (TSource element in source)
    foreach (TResult subElement in selector (element))
      yield return subElement;
}
```

Overview

`SelectMany` concatenates subsequences into a single flat output sequence.

Recall that for each input element, `Select` yields exactly one output element. In contrast, `SelectMany` yields *0..n* output elements. The *0..n* elements come from a subsequence or child sequence that the lambda expression must emit.

You can use `SelectMany` to expand child sequences, flatten nested collections, and join two collections into a flat output sequence. Using the conveyor belt analogy, `SelectMany` funnels fresh material onto a conveyor belt. With `SelectMany`, each input element is the *trigger* for the introduction of fresh material. The fresh material is emitted by the `selector` lambda expression and must be a sequence. In other words, the lambda expression must emit a *child sequence* per input *element*. The final result is a concatenation of the child sequences emitted for each input element.

Starting with a simple example, suppose that we have the following array of names,

```
string[] fullNames = { "Anne Williams", "John Fred Smith", "Sue Green" };
```

that we want to convert to a single flat collection of words—in other words:

```
"Anne", "Williams", "John", "Fred", "Smith", "Sue", Green"
```

`SelectMany` is ideal for this task, because we're mapping each input element to a variable number of output elements. All we must do is come up with a `selector` expression that converts each input element to a child sequence. `string.Split` does the job nicely: it takes a string and splits it into words, emitting the result as an array:

```
string testInputElement = "Anne Williams";
string[] childSequence  = testInputElement.Split();

// childSequence is { "Anne", "Williams" };
```

So, here's our `SelectMany` query and the result:

```
IEnumerable<string> query = fullNames.SelectMany (name => name.Split());

foreach (string name in query)
  Console.Write (name + "|");  // Anne|Williams|John|Fred|Smith|Sue|Green|
```

> **NOTE**
>
> If you replace `SelectMany` with `Select`, you get the same results in hierarchical form. The following emits a sequence of string *arrays*, requiring nested `foreach` statements to enumerate:
>
> ```
> IEnumerable<string[]> query =
>   fullNames.Select (name => name.Split());
>
> foreach (string[] stringArray in query)
>   foreach (string name in stringArray)
>     Console.Write (name + "|");
> ```
>
> The benefit of `SelectMany` is that it yields a single *flat* result sequence.

`SelectMany` is supported in query syntax and is invoked by having an *additional generator*—in other words, an extra `from` clause in the query. The `from` keyword has two meanings in query syntax. At the start of a query, it introduces the original range variable and input sequence. *Anywhere else* in the query, it translates to `SelectMany`. Here's our query in query syntax:

```
IEnumerable<string> query =
  from fullName in fullNames
  from name in fullName.Split()     // Translates to SelectMany
  select name;
```

Note that the additional generator introduces a new range variable—in this case, `name`. The old range variable stays in scope, however, and we can subsequently access both.

Multiple range variables

In the preceding example, both `name` and `fullName` remain in scope until the query either ends or reaches an `into` clause. The extended scope of these variables is *the* killer scenario for query syntax over fluent syntax.

To illustrate, we can take the preceding query and include `fullName` in the final projection:

```
IEnumerable<string> query =
  from fullName in fullNames
  from name in fullName.Split()
  select name + " came from " + fullName;

Anne came from Anne Williams
Williams came from Anne Williams
John came from John Fred Smith
...
```

Behind the scenes, the compiler must pull some tricks to let you access both variables. A good way to appreciate this is to try writing the same query in fluent syntax. It's tricky! It becomes yet more difficult if you insert a `where` or `orderby` clause before projecting:

```
from fullName in fullNames
from name in fullName.Split()
orderby fullName, name
select name + " came from " + fullName;
```

The problem is that `SelectMany` emits a flat sequence of child elements—in our case, a flat collection of words. The original "outer" element from which it came (`fullName`) is lost. The solution is to "carry" the outer element with each child, in a temporary anonymous type:

```
from fullName in fullNames
from x in fullName.Split().Select (name => new { name, fullName } )
orderby x.fullName, x.name
select x.name + " came from " + x.fullName;
```

The only change here is that we're wrapping each child element (`name`) in an anonymous type that also contains its `fullName`. This is similar to how a `let` clause is resolved. Here's the final conversion to fluent syntax:

```
IEnumerable<string> query = fullNames
  .SelectMany (fName => fName.Split()
                             .Select (name => new { name, fName } ))
  .OrderBy (x => x.fName)
  .ThenBy  (x => x.name)
  .Select  (x => x.name + " came from " + x.fName);
```

Thinking in query syntax

As we just demonstrated, there are good reasons to use query syntax if you need multiple range variables. In such cases, it helps to not only use query syntax but also to think directly in its terms.

There are two basic patterns when writing additional generators. The first is *expanding and flattening subsequences*. To do this, you call a property or method on an existing range variable in your additional generator. We did this in the previous example:

```
from fullName in fullNames
from name in fullName.Split()
```

Here, we’ve expanded from enumerating full names to enumerating words. An analogous EF Core query is when you expand collection navigation properties. The following query lists all customers along with their purchases:

```
IEnumerable<string> query = from c in dbContext.Customers
                            from p in c.Purchases
                            select c.Name + " bought a " + p.Description;
```

```
Tom bought a Bike
Tom bought a Holiday
Dick bought a Phone
Harry bought a Car
...
```

Here, we’ve expanded each customer into a subsequence of purchases.

The second pattern is performing a *cartesian product*, or *cross join*, in which every element of one sequence is matched with every element of another. To do this, introduce a generator whose `selector` expression returns a sequence unrelated to a range variable:

```
int[] numbers = { 1, 2, 3 };  string[] letters = { "a", "b" };

IEnumerable<string> query = from n in numbers
                            from l in letters
                            select n.ToString() + l;

// RESULT: { "1a", "1b", "2a", "2b", "3a", "3b" }
```

This style of query is the basis of `SelectMany`-style *joins*.

Joining with SelectMany

You can use `SelectMany` to join two sequences simply by filtering the results of a cross product. For instance, suppose that we want to match players for a game. We could start as follows:

```
string[] players = { "Tom", "Jay", "Mary" };

IEnumerable<string> query = from name1 in players
                            from name2 in players
                            select name1 + " vs " + name2;

//RESULT: { "Tom vs Tom", "Tom vs Jay", "Tom vs Mary",
//          "Jay vs Tom", "Jay vs Jay", "Jay vs Mary",
//          "Mary vs Tom", "Mary vs "Jay", "Mary vs Mary" }
```

The query reads "For every player, reiterate every player, selecting player 1 versus player 2." Although we got what we asked for (a cross join), the results are not useful until we add a filter:

```
IEnumerable<string> query = from name1 in players
                            from name2 in players
                            where name1.CompareTo (name2) < 0
                            orderby name1, name2
                            select name1 + " vs " + name2;

//RESULT: { "Jay vs Mary", "Jay vs Tom", "Mary vs Tom" }
```

The filter predicate constitutes the *join condition*. Our query can be called a *non-equi join* because the join condition doesn't use an equality operator.

SelectMany in EF Core

`SelectMany` in EF Core can perform cross joins, non-equi joins, inner joins, and left outer joins. You can use `SelectMany` with both predefined associations and ad hoc relationships—just as with `Select`. The difference is that `SelectMany` returns a flat rather than a hierarchical result set.

An EF Core cross join is written just as in the preceding section. The following query matches every customer to every purchase (a cross join):

```
var query = from c in dbContext.Customers
            from p in dbContext.Purchases
            select c.Name + " might have bought a " + p.Description;
```

More typically, though, you'd want to match customers to only their own purchases. You achieve this by adding a `where` clause with a joining predicate. This results in a standard SQL-style equi-join:

```
var query = from c in dbContext.Customers
            from p in dbContext.Purchases
            where c.ID == p.CustomerID
            select c.Name + " bought a " + p.Description;
```

> **NOTE**
>
> This translates well to SQL. In the next section, we see how it extends to support outer joins. Reformulating such queries with LINQ's `Join` operator actually makes them *less* extensible—LINQ is opposite to SQL in this sense.

If you have collection navigation properties in your entities, you can express the same query by expanding the subcollection instead of filtering the cross product:

```
from c in dbContext.Customers
from p in c.Purchases
select new { c.Name, p.Description };
```

The advantage is that we've eliminated the joining predicate. We've gone from filtering a cross product to expanding and flattening.

You can add `where` clauses to such a query for additional filtering. For instance, if we want only customers whose names started with "T", we could filter as follows:

```
from c in dbContext.Customers
where c.Name.StartsWith ("T")
from p in c.Purchases
select new { c.Name, p.Description };
```

This EF Core query would work equally well if the `where` clause were moved one line down because the same SQL is generated in both cases. If it is a local query, however, moving the `where` clause down would make it less efficient. With local queries, you should filter *before* joining.

You can introduce new tables into the mix with additional `from` clauses. For instance, if each purchase had purchase item child rows, you could produce a flat result set of customers with their purchases, each with their purchase detail lines as follows:

```
from c in dbContext.Customers
from p in c.Purchases
from pi in p.PurchaseItems
select new { c.Name, p.Description, pi.Detail };
```

Each `from` clause introduces a new *child* table. To include data from a *parent* table (via a navigation property), you don't add a `from` clause—you simply navigate to the property. For example, if each customer has a salesperson whose name you want to query, just do this:

```
from c in dbContext.Customers
select new { Name = c.Name, SalesPerson = c.SalesPerson.Name };
```

You don't use `SelectMany` in this case because there's no subcollection to flatten. Parent navigation properties return a single item.

Outer joins with SelectMany

We saw previously that a `Select` subquery yields a result analogous to a left outer join:

```
from c in dbContext.Customers
select new {
             c.Name,
             Purchases = from p in c.Purchases
                         where p.Price > 1000
                         select new { p.Description, p.Price }
           };
```

In this example, every outer element (customer) is included, regardless of whether the customer has any purchases. But suppose that we rewrite this query with `SelectMany` so that we can obtain a single flat collection rather than a hierarchical result set:

```
from c in dbContext.Customers
from p in c.Purchases
where p.Price > 1000
select new { c.Name, p.Description, p.Price };
```

In the process of flattening the query, we've switched to an inner join: customers are now included only for whom one or more high-value purchases exist. To get a left outer join with a flat result set, we must apply the `DefaultIfEmpty` query operator on the inner sequence. This method returns a sequence with a single null element if its input sequence has no elements. Here's such a query, price predicate aside:

```
from c in dbContext.Customers
from p in c.Purchases.DefaultIfEmpty()
select new { c.Name, p.Description, Price = (decimal?) p.Price };
```

This works perfectly with EF Core, returning all customers—even if they have no purchases. But if we were to run this as a local query, it would crash because when `p` is null, `p.Description` and `p.Price` throw a `NullReferenceException`. We can make our query robust in either scenario, as follows:

```
from c in dbContext.Customers
from p in c.Purchases.DefaultIfEmpty()
select new {
             c.Name,
             Descript = p == null ? null : p.Description,
             Price = p == null ? (decimal?) null : p.Price
           };
```

Let's now reintroduce the price filter. We cannot use a `where` clause as we did before, because it would execute *after* `DefaultIfEmpty`:

```
from c in dbContext.Customers
from p in c.Purchases.DefaultIfEmpty()
where p.Price > 1000...
```

The correct solution is to splice the `Where` clause *before* `DefaultIfEmpty` with a subquery:

```
from c in dbContext.Customers
from p in c.Purchases.Where (p => p.Price > 1000).DefaultIfEmpty()
select new {
             c.Name,
             Descript = p == null ? null : p.Description,
             Price = p == null ? (decimal?) null : p.Price
           };
```

EF Core translates this to a left outer join. This is an effective pattern for writing such queries.

> **NOTE**
>
> If you're used to writing outer joins in SQL, you might be tempted to overlook the simpler option of a `Select` subquery for this style of query in favor of the awkward but familiar SQL-centric flat approach. The hierarchical result set from a `Select` subquery is often better suited to outer join–style queries because there are no additional nulls to deal with.

Joining

Method	Description	SQL equivalents
Join	Applies a lookup strategy to match elements from two collections, emitting a flat result set	INNER JOIN
GroupJoin	Similar to Join, but emits a *hierarchical* result set	INNER JOIN, LEFT OUTER JOIN
Zip	Enumerates two sequences in step (like a zipper), applying a function over each element pair	Exception thrown

Join and GroupJoin

IEnumerable<TOuter>, IEnumerable<TInner>→IEnumerable<TResult>

Join arguments

Argument	Type
Outer sequence	IEnumerable<TOuter>
Inner sequence	IEnumerable<TInner>
Outer key selector	TOuter => TKey
Inner key selector	TInner => TKey
Result selector	(TOuter,TInner) => TResult

GroupJoin arguments

Argument	Type
Outer sequence	IEnumerable<TOuter>
Inner sequence	IEnumerable<TInner>
Outer key selector	TOuter => TKey
Inner key selector	TInner => TKey
Result selector	(TOuter,**IEnumerable<TInner>**) => TResult

Query syntax

```
from outer-var in outer-enumerable
join inner-var in inner-enumerable on outer-key-expr equals inner-key-expr
 [ into identifier ]
```

Overview

Join and GroupJoin mesh two input sequences into a single output sequence. Join emits flat output; GroupJoin emits hierarchical output.

Join and GroupJoin provide an alternative strategy to Select and SelectMany. The advantage of Join and GroupJoin is that they execute efficiently over local in-memory collections because they first load the inner sequence into a keyed lookup, avoiding the need to repeatedly enumerate over every inner element. The disadvantage is that they offer the equivalent of inner and left outer joins only; cross joins and non-equi joins must still be done using Select/SelectMany. With EF Core queries, Join and GroupJoin offer no real benefits over Select and SelectMany.

Table 9-1 summarizes the differences between each of the joining strategies.

Table 9-1. Joining strategies

Strategy	Result shape	Local query efficiency	Inner joins	Left joins
`Select` + `SelectMany`	Flat	Bad	Yes	Yes
`Select` + `Select`	Nested	Bad	Yes	Yes
`Join`	Flat	Good	Yes	—
`GroupJoin`	Nested	Good	Yes	Yes
`GroupJoin` + `SelectMany`	Flat	Good	Yes	Yes

Join

The `Join` operator performs an inner join, emitting a flat output sequence.

The following query lists all customers alongside their purchases without using a navigation property:

```
IQueryable<string> query =
  from c in dbContext.Customers
  join p in dbContext.Purchases on c.ID equals p.CustomerID
  select c.Name + " bought a " + p.Description;
```

The results match what we would get from a `SelectMany`-style query:

```
Tom bought a Bike
Tom bought a Holiday
Dick bought a Phone
Harry bought a Car
```

To see the benefit of `Join` over `SelectMany`, we must convert this to a local query. We can demonstrate this by first copying all customers and purchases to arrays and then querying the arrays:

```
Customer[] customers = dbContext.Customers.ToArray();
Purchase[] purchases = dbContext.Purchases.ToArray();
var slowQuery = from c in customers
                from p in purchases where c.ID == p.CustomerID
                select c.Name + " bought a " + p.Description;

var fastQuery = from c in customers
                join p in purchases on c.ID equals p.CustomerID
                select c.Name + " bought a " + p.Description;
```

Although both queries yield the same results, the `Join` query is considerably faster because its implementation in `Enumerable` preloads the inner collection (`purchases`) into a keyed lookup.

The query syntax for `join` can be written in general terms, as follows:

```
join inner-var in inner-sequence on outer-key-expr equals inner-key-expr
```

Join operators in LINQ differentiate between the *outer sequence* and *inner sequence*. Syntactically:

- The *outer sequence* is the input sequence (`customers`, in this case).
- The *inner sequence* is the new collection you introduce (`purchases`, in this case).

`Join` performs inner joins, meaning customers without purchases are excluded from the output. With inner joins, you can swap the inner and outer sequences in the query and still get the same results:

```
from p in purchases                                // p is now outer
join c in customers on p.CustomerID equals c.ID    // c is now inner
...
```

You can add further `join` clauses to the same query. If each purchase, for instance, has one or more purchase items, you could join the purchase items, as follows:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID          // first join
join pi in purchaseItems on p.ID equals pi.PurchaseID    // second join
...
```

`purchases` acts as the *inner* sequence in the first join and as the *outer* sequence in the second join. You could obtain the same results (inefficiently) using nested `foreach` statements, as follows:

```
foreach (Customer c in customers)
  foreach (Purchase p in purchases)
    if (c.ID == p.CustomerID)
      foreach (PurchaseItem pi in purchaseItems)
        if (p.ID == pi.PurchaseID)
          Console.WriteLine (c.Name + "," + p.Price + "," + pi.Detail);
```

In query syntax, variables from earlier joins remain in scope—just as they do with `SelectMany`-style queries. You're also permitted to insert `where` and `let` clauses in between `join` clauses.

Joining on multiple keys

You can join on multiple keys with anonymous types, as follows:

```
from x in sequenceX
join y in sequenceY on new { K1 = x.Prop1, K2 = x.Prop2 }
                equals new { K1 = y.Prop3, K2 = y.Prop4 }
...
```

For this to work, the two anonymous types must be structured identically. The compiler then implements each with the same internal type, making the joining keys compatible.

Joining in fluent syntax

The following query syntax join

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
select new { c.Name, p.Description, p.Price };
```

in fluent syntax is as follows:

```
customers.Join (                    // outer collection
      purchases,                    // inner collection
      c => c.ID,                    // outer key selector
      p => p.CustomerID,            // inner key selector
      (c, p) => new
         { c.Name, p.Description, p.Price }    // result selector
);
```

The result selector expression at the end creates each element in the output sequence. If you have additional clauses prior to projecting, such as `orderby` in this example:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
orderby p.Price
select c.Name + " bought a " + p.Description;
```

you must manufacture a temporary anonymous type in the result selector in fluent syntax. This keeps both `c` and `p` in scope following the join:

```
customers.Join (                    // outer collection
      purchases,                    // inner collection
      c => c.ID,                    // outer key selector
      p => p.CustomerID,            // inner key selector
      (c, p) => new { c, p } )      // result selector
  .OrderBy (x => x.p.Price)
  .Select  (x => x.c.Name + " bought a " + x.p.Description);
```

Query syntax is usually preferable when joining; it's less fiddly.

GroupJoin

`GroupJoin` does the same work as `Join`, but instead of yielding a flat result, it yields a hierarchical result, grouped by each outer element. It also allows left outer joins. `GroupJoin` is not currently supported in EF Core.

The query syntax for `GroupJoin` is the same as for `Join`, but is followed by the `into` keyword.

Here's the most basic example, using a local query:

```
Customer[] customers = dbContext.Customers.ToArray();
Purchase[] purchases = dbContext.Purchases.ToArray();

IEnumerable<IEnumerable<Purchase>> query =
  from c in customers
  join p in purchases on c.ID equals p.CustomerID
  into custPurchases
  select custPurchases;   // custPurchases is a sequence
```

NOTE

An `into` clause translates to `GroupJoin` only when it appears directly after a `join` clause. After a `select` or `group` clause, it means *query continuation*. The two uses of the `into` keyword are quite different, although they have one feature in common: they both introduce a new range variable.

The result is a sequence of sequences, which we could enumerate as follows:

```
foreach (IEnumerable<Purchase> purchaseSequence in query)
  foreach (Purchase p in purchaseSequence)
    Console.WriteLine (p.Description);
```

This isn't very useful, however, because `purchaseSequence` has no reference to the customer. More commonly, you'd do this:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
into custPurchases
select new { CustName = c.Name, custPurchases };
```

This gives the same results as the following (inefficient) `Select` subquery:

```
from c in customers
select new
{
  CustName = c.Name,
  custPurchases = purchases.Where (p => c.ID == p.CustomerID)
};
```

By default, `GroupJoin` does the equivalent of a left outer join. To get an inner join—whereby customers without purchases are excluded—you need to filter on `custPurchases`:

```
from c in customers join p in purchases on c.ID equals p.CustomerID
into custPurchases
where custPurchases.Any()
select ...
```

Clauses after a group-join `into` operate on *subsequences* of inner child elements, not *individual* child elements. This means that to filter individual purchases, you'd need to call `Where` *before* joining:

```
from c in customers
join p in purchases.Where (p2 => p2.Price > 1000)
  on c.ID equals p.CustomerID
into custPurchases ...
```

You can construct lambda queries with `GroupJoin` as you would with `Join`.

Flat outer joins

You run into a dilemma if you want both an outer join and a flat result set. `GroupJoin` gives you the outer join; `Join` gives you the flat result set. The solution is to first call `GroupJoin`, then `DefaultIfEmpty` on each child sequence, and then finally `SelectMany` on the result:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID into custPurchases
from cp in custPurchases.DefaultIfEmpty()
select new
{
  CustName = c.Name,
  Price = cp == null ? (decimal?) null : cp.Price
};
```

`DefaultIfEmpty` emits a sequence with a single null value if a subsequence of purchases is empty. The second `from` clause translates to `SelectMany`. In this role, it *expands and flattens* all the purchase subsequences, concatenating them into a single sequence of purchase *elements*.

Joining with lookups

The `Join` and `GroupJoin` methods in `Enumerable` work in two steps. First, they load the inner sequence into a *lookup*. Second, they query the outer sequence in combination with the lookup.

A *lookup* is a sequence of groupings that can be accessed directly by key. Another way to think of it is as a dictionary of sequences—a dictionary that can accept many elements under each key (sometimes called a *multidictionary*). Lookups are read-only and defined by the following interface:

```
public interface ILookup<TKey,TElement> :
   IEnumerable<IGrouping<TKey,TElement>>, IEnumerable
{
  int Count { get; }
  bool Contains (TKey key);
  IEnumerable<TElement> this [TKey key] { get; }
}
```

> **NOTE**
>
> The joining operators—like other sequence-emitting operators—honor deferred or lazy execution semantics. This means the lookup is not built until you begin enumerating the output sequence (and then the *entire* lookup is built right then).

You can create and query lookups manually as an alternative strategy to using the joining operators when dealing with local collections. There are a couple of benefits to doing so:

- You can reuse the same lookup over multiple queries—as well as in ordinary imperative code.
- Querying a lookup is an excellent way of understanding how `Join` and `GroupJoin` work.

The `ToLookup` extension method creates a lookup. The following loads all purchases into a lookup—keyed by their `CustomerID`:

```
ILookup<int?,Purchase> purchLookup =
  purchases.ToLookup (p => p.CustomerID, p => p);
```

The first argument selects the key; the second argument selects the objects that are to be loaded as values into the lookup.

Reading a lookup is rather like reading a dictionary except that the indexer returns a *sequence* of matching items rather than a *single* matching item. The following enumerates all purchases made by the customer whose ID is 1:

```
foreach (Purchase p in purchLookup [1])
  Console.WriteLine (p.Description);
```

With a lookup in place, you can write `SelectMany`/`Select` queries that execute as efficiently as `Join`/`GroupJoin` queries. `Join` is equivalent to using `SelectMany` on a lookup:

```
from c in customers
from p in purchLookup [c.ID]
select new { c.Name, p.Description, p.Price };

Tom Bike 500
Tom Holiday 2000
Dick Bike 600
Dick Phone 300
...
```

Adding a call to `DefaultIfEmpty` makes this into an outer join:

```
from c in customers
from p in purchLookup [c.ID].DefaultIfEmpty()
 select new {
             c.Name,
             Descript = p == null ? null : p.Description,
             Price = p == null ? (decimal?) null : p.Price
           };
```

`GroupJoin` is equivalent to reading the lookup inside a projection:

```
from c in customers
select new {
             CustName = c.Name,
             CustPurchases = purchLookup [c.ID]
           };
```

Enumerable implementations

Here's the simplest valid implementation of `Enumerable.Join`, null checking aside:

```
public static IEnumerable <TResult> Join
                                    <TOuter,TInner,TKey,TResult> (
  this IEnumerable <TOuter>     outer,
  IEnumerable <TInner>          inner,
```

```
  Func <TOuter,TKey>           outerKeySelector,
  Func <TInner,TKey>           innerKeySelector,
  Func <TOuter,TInner,TResult>  resultSelector)
{
  ILookup <TKey, TInner> lookup = inner.ToLookup (innerKeySelector);
  return
    from outerItem in outer
    from innerItem in lookup [outerKeySelector (outerItem)]
    select resultSelector (outerItem, innerItem);
}
```

`GroupJoin`'s implementation is like that of `Join` but simpler:

```
public static IEnumerable <TResult> GroupJoin
                                    <TOuter,TInner,TKey,TResult> (
  this IEnumerable <TOuter>     outer,
  IEnumerable <TInner>          inner,
  Func <TOuter,TKey>            outerKeySelector,
  Func <TInner,TKey>            innerKeySelector,
  Func <TOuter,IEnumerable<TInner>,TResult>  resultSelector)
{
  ILookup <TKey, TInner> lookup = inner.ToLookup (innerKeySelector);
  return
    from outerItem in outer
    select resultSelector
     (outerItem, lookup [outerKeySelector (outerItem)]);
}
```

The Zip Operator

`IEnumerable<TFirst>`,
`IEnumerable<TSecond>`→`IEnumerable<TResult>`

The `Zip` operator enumerates two sequences in step (like a zipper), returning a sequence based on applying a function over each element pair. For instance, the following:

```
int[] numbers = { 3, 5, 7 };
string[] words = { "three", "five", "seven", "ignored" };
IEnumerable<string> zip = numbers.Zip (words, (n, w) => n + "=" + w);
```

produces a sequence with the following elements:

```
3=three
5=five
7=seven
```

Extra elements in either input sequence are ignored. `Zip` is not supported by EF Core.

Ordering

`IEnumerable<TSource>→IOrderedEnumerable<TSource>`

Method	Description	SQL equivalents
`OrderBy, ThenBy`	Sorts a sequence in ascending order	`ORDER BY ...`
`OrderByDescending, ThenByDescending`	Sorts a sequence in descending order	`ORDER BY ... DESC`
`Reverse`	Returns a sequence in reverse order	Exception thrown

Ordering operators return the same elements in a different order.

OrderBy, OrderByDescending, ThenBy, ThenByDescending

OrderBy and OrderByDescending arguments

Argument	Type
Input sequence	`IEnumerable<TSource>`
Key selector	`TSource => TKey`

Return type = `IOrderedEnumerable<TSource>`

ThenBy and ThenByDescending arguments

Argument	Type
Input sequence	`IOrderedEnumerable<TSource>`
Key selector	`TSource => TKey`

Query syntax

```
orderby expression1 [descending] [, expression2 [descending] ... ]
```

Overview

`OrderBy` returns a sorted version of the input sequence, using the `keySelector` expression to make comparisons. The following query emits a sequence of names in alphabetical order:

```
IEnumerable<string> query = names.OrderBy (s => s);
```

The following sorts names by length:

```
IEnumerable<string> query = names.OrderBy (s => s.Length);

// Result: { "Jay", "Tom", "Mary", "Dick", "Harry" };
```

The relative order of elements with the same sorting key (in this case, Jay/Tom and Mary/Dick) is indeterminate—unless you append a `ThenBy` operator:

```
IEnumerable<string> query = names.OrderBy (s => s.Length).ThenBy (s => s);

// Result: { "Jay", "Tom", "Dick", "Mary", "Harry" };
```

`ThenBy` reorders only elements that had the same sorting key in the preceding sort. You can chain any number of `ThenBy` operators. The following sorts first by length, then by the second character, and finally by the first character:

```
names.OrderBy (s => s.Length).ThenBy (s => s[1]).ThenBy (s => s[0]);
```

Here's the equivalent in query syntax:

```
from s in names
orderby s.Length, s[1], s[0]
select s;
```

WARNING

The following variation is *incorrect*—it will actually order first by `s[1]` and then by `s.Length` (or in the case of a database query, it will order *only* by `s[1]` and discard the former ordering):

```
from s in names
orderby s.Length
orderby s[1]
...
```

LINQ also provides `OrderByDescending` and `ThenByDescending` operators, which do the same things, emitting the results in reverse order. The following EF Core query retrieves purchases in descending order of price, with those of the same price listed alphabetically:

```
dbContext.Purchases.OrderByDescending (p => p.Price)
                    .ThenBy (p => p.Description);
```

In query syntax:

```
from p in dbContext.Purchases
orderby p.Price descending, p.Description
select p;
```

Comparers and collations

In a local query, the key selector objects themselves determine the ordering algorithm via their default `IComparable` implementation (see Chapter 7). You can override the sorting algorithm by passing in an `IComparer` object. The following performs a case-insensitive sort:

```
names.OrderBy (n => n, StringComparer.CurrentCultureIgnoreCase);
```

Passing in a comparer is not supported in query syntax or in any way by EF Core. When querying a database, the comparison algorithm is determined by the participating column's collation. If the collation is case sensitive, you can request a case-insensitive sort by calling `ToUpper` in the key selector:

```
from p in dbContext.Purchases
orderby p.Description.ToUpper()
select p;
```

IOrderedEnumerable and IOrderedQueryable

The ordering operators return special subtypes of `IEnumerable<T>`. Those in `Enumerable` return `IOrderedEnumerable<TSource>`; those in `Queryable` return `IOrderedQueryable<TSource>`. These subtypes allow a subsequent `ThenBy` operator to refine rather than replace the existing ordering.

The additional members that these subtypes define are not publicly exposed, so they present like ordinary sequences. The fact that they are different types comes into play when building queries progressively:

```
IOrderedEnumerable<string> query1 = names.OrderBy (s => s.Length);
IOrderedEnumerable<string> query2 = query1.ThenBy (s => s);
```

If we instead declare `query1` of type `IEnumerable<string>`, the second line would not compile—`ThenBy` requires an input of type `IOrderedEnumerable<string>`. You can avoid worrying about this by implicitly typing range variables:

```
var query1 = names.OrderBy (s => s.Length);
var query2 = query1.ThenBy (s => s);
```

Implicit typing can create problems of its own, though. The following will not compile:

```
var query = names.OrderBy (s => s.Length);
query = query.Where (n => n.Length > 3);       // Compile-time error
```

The compiler infers `query` to be of type `IOrderedEnumerable<string>`, based on `OrderBy`'s output sequence type. However, the `Where` on the next line returns an ordinary `IEnumerable<string>`, which cannot be assigned back to `query`. You can work around this either with explicit typing or by calling `AsEnumerable()` after `OrderBy`:

```
var query = names.OrderBy (s => s.Length).AsEnumerable();
query = query.Where (n => n.Length > 3);                   // OK
```

The equivalent in interpreted queries is to call `AsQueryable`.

Grouping

Method	Description	SQL equivalents
`GroupBy`	Groups a sequence into subsequences	`GROUP BY`
`Chunk`	Groups a sequence into arrays of a fixed size	

GroupBy

`IEnumerable<TSource>→IEnumerable<IGrouping<TKey,TElement>>`

Argument	Type
Input sequence	`IEnumerable<TSource>`
Key selector	`TSource => TKey`
Element selector (optional)	`TSource => TElement`
Comparer (optional)	`IEqualityComparer<TKey>`

Query syntax

```
group element-expression by key-expression
```

Overview

`GroupBy` organizes a flat input sequence into sequences of *groups*. For example, the following organizes all of the files in *Path.GetTempPath()* by extension:

```
string[] files = Directory.GetFiles (Path.GetTempPath());

IEnumerable<IGrouping<string,string>> query =
  files.GroupBy (file => Path.GetExtension (file));
```

Or, with implicit typing:

```
var query = files.GroupBy (file => Path.GetExtension (file));
```

Here’s how to enumerate the result:

```
foreach (IGrouping<string,string> grouping in query)
{
  Console.WriteLine ("Extension: " + grouping.Key);
  foreach (string filename in grouping)
    Console.WriteLine ("   - " + filename);
}

Extension: .pdf
  -- chapter03.pdf
  -- chapter04.pdf
Extension: .doc
  -- todo.doc
  -- menu.doc
  -- Copy of menu.doc
...
```

`Enumerable.GroupBy` works by reading the input elements into a temporary dictionary of lists so that all elements with the same key end up in the same sublist. It then emits a sequence of *groupings*. A grouping is a sequence with a `Key` property:

```
public interface IGrouping <TKey,TElement> : IEnumerable<TElement>,
                                             IEnumerable
{
  TKey Key { get; }    // Key applies to the subsequence as a whole
}
```

By default, the elements in each grouping are untransformed input elements unless you specify an `elementSelector` argument. The following projects each input element to uppercase:

```
files.GroupBy (file => Path.GetExtension (file), file => file.ToUpper());
```

An `elementSelector` is independent of the `keySelector`. In our case, this means that the `Key` on each grouping is still in its original case:

```
Extension: .pdf
  -- CHAPTER03.PDF
  -- CHAPTER04.PDF
Extension: .doc
  -- TODO.DOC
```

Note that the subcollections are not emitted in alphabetical order of key. `GroupBy` merely *groups;* it does not *sort*. In fact, it preserves the original ordering. To sort, you must add an `OrderBy` operator:

```
files.GroupBy (file => Path.GetExtension (file), file => file.ToUpper())
     .OrderBy (grouping => grouping.Key);
```

`GroupBy` has a simple and direct translation in query syntax:

```
group element-expr by key-expr
```

Here's our example in query syntax:

```
from file in files
group file.ToUpper() by Path.GetExtension (file);
```

As with `select`, `group` “ends” a query—unless you add a query continuation clause:

```
from file in files
group file.ToUpper() by Path.GetExtension (file) into grouping
orderby grouping.Key
select grouping;
```

Query continuations are often useful in a `group by` query. The next query filters out groups that have fewer than five files in them:

```
from file in files
group file.ToUpper() by Path.GetExtension (file) into grouping
where grouping.Count() >= 5
select grouping;
```

> **NOTE**
>
> A `where` after a `group by` is equivalent to HAVING in SQL. It applies to each subsequence or grouping as a whole rather than the individual elements.

Sometimes, you’re interested purely in the result of an aggregation on a grouping and so can abandon the subsequences:

```
string[] votes = { "Dogs", "Cats", "Cats", "Dogs", "Dogs" };

IEnumerable<string> query = from vote in votes
                            group vote by vote into g
                            orderby g.Count() descending
                            select g.Key;

string winner = query.First();    // Dogs
```

GroupBy in EF Core

Grouping works in the same way when querying a database. If you have navigation properties set up, you'll find, however, that the need to group arises less frequently than with standard SQL. For instance, to select customers with at least two purchases, you don't need to `group`; the following query does the job nicely:

```
from c in dbContext.Customers
where c.Purchases.Count >= 2
select c.Name + " has made " + c.Purchases.Count + " purchases";
```

An example of when you might use grouping is to list total sales by year:

```
from p in dbContext.Purchases
group p.Price by p.Date.Year into salesByYear
select new {
             Year       = salesByYear.Key,
             TotalValue = salesByYear.Sum()
           };
```

LINQ's grouping is more powerful than SQL's GROUP BY in that you can fetch all detail rows without any aggregation:

```
from p in dbContext.Purchases
group p by p.Date.Year
Date.Year
```

However, this doesn't work in EF Core. An easy workaround is to call `.AsEnumerable()` just before grouping so that the grouping happens on the client. This is no less efficient as long as you perform any filtering *before* grouping so that you only fetch the data you need from the server.

Another departure from traditional SQL comes in there being no obligation to project the variables or expressions used in grouping or sorting.

Grouping by multiple keys

You can group by a composite key, using an anonymous type:

```
from n in names
group n by new { FirstLetter = n[0], Length = n.Length };
```

Custom equality comparers

You can pass a custom equality comparer into `GroupBy`, in a local query, to change the algorithm for key comparison. Rarely is this required, though, because changing the key selector expression is usually sufficient. For instance, the following creates a case-insensitive grouping:

```
group n by n.ToUpper()
```

Chunk

`IEnumerable<TSource>`→`IEnumerable<TElement[]>`

Argument	Type
Input sequence	`IEnumerable<TSource>`
`size`	`int`

Introduced in .NET 6, `Chunk` groups a sequence into chunks of a given size (or fewer, if there aren't enough elements):

```
foreach (int[] chunk in new[] { 1, 2, 3, 4, 5, 6, 7, 8 }.Chunk (3))
  Console.WriteLine (string.Join (", ", chunk));
```

Output:

```
1, 2, 3
4, 5, 6
7, 8
```

Set Operators

`IEnumerable<TSource>, IEnumerable<TSource>→IEnumerable<TSource>`

Method	Description	SQL equivalents
`Concat`	Returns a concatenation of elements in each of the two sequences	`UNION ALL`
`Union`, `UnionBy`	Returns a concatenation of elements in each of the two sequences, excluding duplicates	`UNION`
`Intersect`, `IntersectBy`	Returns elements present in both sequences	`WHERE ... IN (...)`
`Except`, `ExceptBy`	Returns elements present in the first but not the second sequence	`EXCEPT` *or* `WHERE ... NOT IN (...)`

Concat, Union, UnionBy

`Concat` returns all the elements of the first sequence, followed by all the elements of the second. `Union` does the same but removes any duplicates:

```
int[] seq1 = { 1, 2, 3 }, seq2 = { 3, 4, 5 };

IEnumerable<int>
  concat = seq1.Concat (seq2),    //  { 1, 2, 3, 3, 4, 5 }
  union  = seq1.Union  (seq2);    //  { 1, 2, 3, 4, 5 }
```

Specifying the type argument explicitly is useful when the sequences are differently typed but the elements have a common base type. For instance, with the reflection API (Chapter 18), methods and properties are represented with `MethodInfo` and `PropertyInfo` classes, which have a common base class called `MemberInfo`. We can concatenate methods and properties by stating that base class explicitly when calling `Concat`:

```
MethodInfo[] methods = typeof (string).GetMethods();
PropertyInfo[] props = typeof (string).GetProperties();
IEnumerable<MemberInfo> both = methods.Concat<MemberInfo> (props);
```

In the next example, we filter the methods before concatenating:

```
var methods = typeof (string).GetMethods().Where (m => !m.IsSpecialName);
var props = typeof (string).GetProperties();
var both = methods.Concat<MemberInfo> (props);
```

This example relies on interface type parameter variance: `methods` is of type `IEnumerable<MethodInfo>`, which requires a covariant conversion to `IEnumerable<MemberInfo>`. It's a good illustration of how variance makes things work more like you'd expect.

`UnionBy` (introduced in .NET 6) takes a `keySelector`, which is used in determining whether an element is a duplicate. In the following example, we perform a case-insensitive union:

```
string[] seq1 = { "A", "b", "C" };
string[] seq2 = { "a", "B", "c" };
var union = seq1.UnionBy (seq2, x => x.ToUpperInvariant());
// union is { "A", "b", "C" }
```

In this case, the same thing can be accomplished with `Union`, if we supply an equality comparer:

```
var union = seq1.Union (seq2, StringComparer.InvariantCultureIgnoreCase);
```

Intersect, Intersect By, Except, and ExceptBy

`Intersect` returns the elements that two sequences have in common. `Except` returns the elements in the first input sequence that are *not* present in the second:

```
int[] seq1 = { 1, 2, 3 }, seq2 = { 3, 4, 5 };

IEnumerable<int>
  commonality = seq1.Intersect (seq2),    //  { 3 }
  difference1 = seq1.Except    (seq2),    //  { 1, 2 }
  difference2 = seq2.Except    (seq1);    //  { 4, 5 }
```

`Enumerable.Except` works internally by loading all of the elements in the first collection into a dictionary and then removing from the dictionary all elements present in the second sequence. The equivalent in SQL is a `NOT EXISTS` or `NOT IN` subquery:

```
SELECT number FROM numbers1Table
WHERE number NOT IN (SELECT number FROM numbers2Table)
```

The `IntersectBy` and `ExceptBy` methods (from .NET 6) let you specify a key selector that's applied before performing equality comparison (see the discussion on `UnionBy` in the preceding section).

Conversion Methods

LINQ deals primarily in sequences; in other words, collections of type `IEnumerable<T>`. The conversion methods convert to and from other types of collections:

Method	Description
`OfType`	Converts `IEnumerable` to `IEnumerable<T>`, discarding wrongly typed elements
`Cast`	Converts `IEnumerable` to `IEnumerable<T>`, throwing an exception if there are any wrongly typed elements
`ToArray`	Converts `IEnumerable<T>` to `T[]`
`ToList`	Converts `IEnumerable<T>` to `List<T>`
`ToDictionary`	Converts `IEnumerable<T>` to `Dictionary<TKey,TValue>`
`ToLookup`	Converts `IEnumerable<T>` to `ILookup<TKey,TElement>`
`AsEnumerable`	Upcasts to `IEnumerable<T>`
`AsQueryable`	Casts or converts to `IQueryable<T>`

OfType and Cast

`OfType` and `Cast` accept a nongeneric `IEnumerable` collection and emit a generic `IEnumerable<T>` sequence that you can subsequently query:

```
ArrayList classicList = new ArrayList();          // in System.Collections
classicList.AddRange ( new int[] { 3, 4, 5 } );
IEnumerable<int> sequence1 = classicList.Cast<int>();
```

`Cast` and `OfType` differ in their behavior when encountering an input element that's of an incompatible type. `Cast` throws an exception; `OfType` ignores the incompatible element. Continuing the preceding example:

```
DateTime offender = DateTime.Now;
classicList.Add (offender);
```

```
IEnumerable<int>
  sequence2 = classicList.OfType<int>(), // OK - ignores offending DateTime
  sequence3 = classicList.Cast<int>();   // Throws exception
```

The rules for element compatibility exactly follow those of C#'s `is` operator, and therefore consider only reference conversions and unboxing conversions. We can see this by examining the internal implementation of `OfType`:

```
public static IEnumerable<TSource> OfType <TSource> (IEnumerable source)
{
  foreach (object element in source)
    if (element is TSource)
      yield return (TSource)element;
}
```

`Cast` has an identical implementation, except that it omits the type compatibility test:

```
public static IEnumerable<TSource> Cast <TSource> (IEnumerable source)
{
  foreach (object element in source)
    yield return (TSource)element;
}
```

A consequence of these implementations is that you cannot use `Cast` to perform numeric or custom conversions (for these, you must perform a `Select` operation instead). In other words, `Cast` is not as flexible as C#'s cast operator:

```
int i = 3;
long l = i;         // Implicit numeric conversion int->long
int i2 = (int) l;   // Explicit numeric conversion long->int
```

We can demonstrate this by attempting to use `OfType` or `Cast` to convert a sequence of `int`s to a sequence of `long`s:

```
int[] integers = { 1, 2, 3 };

IEnumerable<long> test1 = integers.OfType<long>();
IEnumerable<long> test2 = integers.Cast<long>();
```

When enumerated, `test1` emits zero elements and `test2` throws an exception. Examining `OfType`'s implementation, it's fairly clear why. After substituting `TSource`, we get the following expression:

```
(element is long)
```

This returns `false` for an `int element`, due to the lack of an inheritance relationship.

> **NOTE**
>
> The reason that `test2` throws an exception when enumerated is more subtle. Notice in `Cast`'s implementation that `element` is of type `object`. When `TSource` is a value type, the CLR assumes this is an *unboxing conversion* and synthesizes a method that reproduces the scenario described in the section "Boxing and Unboxing":
>
> ```
> int value = 123;
> object element = value;
> long result = (long) element;  // exception
> ```
>
> Because the `element` variable is declared of type `object`, an `object`-to-`long` cast is performed (an unboxing) rather than an `int`-to-`long` numeric conversion. Unboxing operations require an exact type match, so the `object`-to-`long` unbox fails when given an `int`.

As we suggested previously, the solution is to use an ordinary `Select`:

```
IEnumerable<long> castLong = integers.Select (s => (long) s);
```

`OfType` and `Cast` are also useful in downcasting elements in a generic input sequence. For instance, if you have an input sequence of type `IEnumerable<Fruit>`, `OfType<Apple>` would return just the apples. This is particularly useful in LINQ to XML (see Chapter 10).

`Cast` has query syntax support: simply precede the range variable with a type:

```
from TreeNode node in myTreeView.Nodes
...
```

ToArray, ToList, ToDictionary, ToHashSet, ToLookup

`ToArray`, `ToList`, and `ToHashSet` emit the results into an array, `List<T>` or `HashSet<T>`. When they execute, these operators force the immediate enumeration of the input sequence. For examples, refer to "Deferred Execution".

`ToDictionary` and `ToLookup` accept the following arguments:

Argument	Type
Input sequence	`IEnumerable<TSource>`
Key selector	`TSource => TKey`
Element selector (optional)	`TSource => TElement`
Comparer (optional)	`IEqualityComparer<TKey>`

`ToDictionary` also forces immediate execution of a sequence, writing the results to a generic `Dictionary`. The `keySelector` expression you provide must evaluate to a unique value for each element in the input sequence; otherwise, an exception is thrown. In contrast, `ToLookup` allows many elements of the same key. We described lookups in "Joining with lookups".

AsEnumerable and AsQueryable

`AsEnumerable` upcasts a sequence to `IEnumerable<T>`, forcing the compiler to bind subsequent query operators to methods in `Enumerable` instead of `Queryable`. For an example, see "Combining Interpreted and Local Queries".

`AsQueryable` downcasts a sequence to `IQueryable<T>` if it implements that interface. Otherwise, it instantiates an `IQueryable<T>` wrapper over the local query.

Element Operators

`IEnumerable<TSource>`→ `TSource`

Method	Description	SQL equivalents
`First`, `FirstOrDefault`	Returns the first element in the sequence, optionally satisfying a predicate	`SELECT TOP 1 ... ORDER BY ...`
`Last`, `LastOrDefault`	Returns the last element in the sequence, optionally satisfying a predicate	`SELECT TOP 1 ... ORDER BY ... DESC`
`Single`, `SingleOrDefault`	Equivalent to `First`/`FirstOrDefault`, but throws an exception if there is more than one match	
`ElementAt`, `ElementAtOrDefault`	Returns the element at the specified position	Exception thrown
`MinBy`, `MaxBy`	Returns the element with the smallest or largest value	Exception thrown
`DefaultIfEmpty`	Returns a single-element sequence whose value is `default(TSource)` if the sequence has no elements	`OUTER JOIN`

Methods ending in "OrDefault" return `default(TSource)` rather than throwing an exception if the input sequence is empty or if no elements match the supplied predicate.

`default(TSource)` is `null` for reference type elements, `false` for the `bool` type, and zero for numeric types.

First, Last, and Single

Argument	Type
Source sequence	`IEnumerable<TSource>`
Predicate (optional)	`TSource => bool`

The following example demonstrates `First` and `Last`:

```
int[] numbers  = { 1, 2, 3, 4, 5 };
int first      = numbers.First();                      // 1
int last       = numbers.Last();                       // 5
int firstEven  = numbers.First  (n => n % 2 == 0);     // 2
int lastEven   = numbers.Last   (n => n % 2 == 0);     // 4
```

The following demonstrates `First` versus `FirstOrDefault`:

```
int firstBigError  = numbers.First          (n => n > 10);   // Exception
int firstBigNumber = numbers.FirstOrDefault (n => n > 10);   // 0
```

To prevent an exception, `Single` requires exactly one matching element; `SingleOrDefault` requires one *or zero* matching elements:

```
int onlyDivBy3 = numbers.Single (n => n % 3 == 0);   // 3
int divBy2Err  = numbers.Single (n => n % 2 == 0);   // Error: 2 & 4 match

int singleError = numbers.Single          (n => n > 10);      // Error
int noMatches   = numbers.SingleOrDefault (n => n > 10);      // 0
int divBy2Error = numbers.SingleOrDefault (n => n % 2 == 0);  // Error
```

`Single` is the “fussiest” in this family of element operators. `FirstOrDefault` and `LastOrDefault` are the most tolerant.

In EF Core, `Single` is often used to retrieve a row from a table by primary key:

```
Customer cust = dataContext.Customers.Single (c => c.ID == 3);
```

ElementAt

Argument	Type
Source sequence	`IEnumerable<TSource>`
Index of element to return	`int`

`ElementAt` picks the *n*th element from the sequence:

```
int[] numbers  = { 1, 2, 3, 4, 5 };
int third      = numbers.ElementAt (2);            // 3
int tenthError = numbers.ElementAt (9);            // Exception
int tenth      = numbers.ElementAtOrDefault (9);   // 0
```

`Enumerable.ElementAt` is written such that if the input sequence happens to implement `IList<T>`, it calls `IList<T>`'s indexer. Otherwise, it enumerates *n* times and then returns the next element. `ElementAt` is not supported in EF Core.

MinBy and MaxBy

`MinBy` and `MaxBy` (introduced in .NET 6) return the element with the smallest or largest value, as determined by a `keySelector`:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
Console.WriteLine (names.MaxBy (n => n.Length));   // Harry
```

In contrast, `Min` and `Max` (which we will cover in the following section) return the smallest or largest value itself:

```
Console.WriteLine (names.Max   (n => n.Length));   // 5
```

If two or more elements share a minimum/maximum value, `MinBy`/`MaxBy` returns the first:

```
Console.WriteLine (names.MinBy (n => n.Length));   // Tom
```

If the input sequence is empty, `MinBy` and `MaxBy` return null if the element type is nullable (or throw an exception if the element type is not nullable).

DefaultIfEmpty

`DefaultIfEmpty` returns a sequence containing a single element whose value is `default(TSource)` if the input sequence has no elements; otherwise, it returns the input sequence unchanged. You use this in writing flat outer joins: see "Outer joins with SelectMany" and "Flat outer joins".

Aggregation Methods

`IEnumerable<TSource>`→*`scalar`*

Method	Description	SQL equivalents
`Count`, `LongCount`	Returns the number of elements in the input sequence, optionally satisfying a predicate	`COUNT (...)`
`Min`, `Max`	Returns the smallest or largest element in the sequence	`MIN (...)`, `MAX (...)`
`Sum`, `Average`	Calculates a numeric sum or average over elements in the sequence	`SUM (...)`, `AVG (...)`
`Aggregate`	Performs a custom aggregation	Exception thrown

Count and LongCount

Argument	Type
Source sequence	`IEnumerable<TSource>`
Predicate (optional)	`TSource => bool`

`Count` simply enumerates over a sequence, returning the number of items:

```
int fullCount = new int[] { 5, 6, 7 }.Count();    // 3
```

The internal implementation of `Enumerable.Count` tests the input sequence to see whether it happens to implement `ICollection<T>`. If it does, it simply calls `ICollection<T>.Count`; otherwise, it enumerates over every item, incrementing a counter.

You can optionally supply a predicate:

```
int digitCount = "pa55w0rd".Count (c => char.IsDigit (c));   // 3
```

`LongCount` does the same job as `Count` but returns a 64-bit integer, allowing for sequences of greater than two billion elements.

Min and Max

Argument	Type
Source sequence	`IEnumerable<TSource>`
Result selector (optional)	`TSource => TResult`

`Min` and `Max` return the smallest or largest element from a sequence:

```
int[] numbers = { 28, 32, 14 };
int smallest = numbers.Min();  // 14;
int largest  = numbers.Max();  // 32;
```

If you include a `selector` expression, each element is first projected:

```
int smallest = numbers.Max (n => n % 10);  // 8;
```

A `selector` expression is mandatory if the items themselves are not intrinsically comparable—in other words, if they do not implement `IComparable<T>`:

```
Purchase runtimeError = dbContext.Purchases.Min ();             // Error
decimal? lowestPrice = dbContext.Purchases.Min (p => p.Price);  // OK
```

A `selector` expression determines not only how elements are compared, but also the final result. In the preceding example, the final result is a decimal value, not a purchase object. To get the cheapest purchase, you need a subquery:

```
Purchase cheapest = dbContext.Purchases
  .Where (p => p.Price == dbContext.Purchases.Min (p2 => p2.Price))
  .FirstOrDefault();
```

In this case, you could also formulate the query without an aggregation by using an `OrderBy` followed by `FirstOrDefault`.

Sum and Average

Argument	Type
Source sequence	`IEnumerable<TSource>`
Result selector (optional)	`TSource => TResult`

`Sum` and `Average` are aggregation operators that are used in a similar manner to `Min` and `Max`:

```
decimal[] numbers  = { 3, 4, 8 };
decimal sumTotal   = numbers.Sum();               // 15
decimal average    = numbers.Average();           // 5   (mean value)
```

The following returns the total length of each of the strings in the `names` array:

```
int combinedLength = names.Sum (s => s.Length);   // 19
```

`Sum` and `Average` are fairly restrictive in their typing. Their definitions are hardwired to each of the numeric types (`int`, `long`, `float`, `double`, `decimal`, and their nullable versions). In contrast, `Min` and `Max` can operate directly on anything that implements `IComparable<T>`—such as a `string`, for instance.

Further, `Average` always returns either `decimal`, `float`, or `double`, according to the following table:

Selector type	Result type
`decimal`	`decimal`
`float`	`float`
`int, long, double`	`double`

This means that the following does not compile ("cannot convert `double` to `int`"):

```
int avg = new int[] { 3, 4 }.Average();
```

But this will compile:

```
double avg = new int[] { 3, 4 }.Average();   // 3.5
```

`Average` implicitly upscales the input values to prevent loss of precision. In this example, we averaged integers and got 3.5 without needing to resort to an input element cast:

```
double avg = numbers.Average (n => (double) n);
```

When querying a database, `Sum` and `Average` translate to the standard SQL aggregations. The following query returns customers whose average purchase was more than $500:

```
from c in dbContext.Customers
where c.Purchases.Average (p => p.Price) > 500
select c.Name;
```

Aggregate

`Aggregate` allows you to specify a custom accumulation algorithm for implementing unusual aggregations. `Aggregate` is not supported in EF Core and is somewhat specialized in its use cases. The following demonstrates how `Aggregate` can do the work of `Sum`:

```
int[] numbers = { 1, 2, 3 };
int sum = numbers.Aggregate (0, (total, n) => total + n);   // 6
```

The first argument to `Aggregate` is the *seed*, from which accumulation starts. The second argument is an expression to update the accumulated value, given a fresh element. You can optionally supply a third argument to project the final result value from the accumulated value.

> **NOTE**
>
> Most problems for which `Aggregate` has been designed can be solved as easily with a `foreach` loop—and with more familiar syntax. The advantage of using `Aggregate` is that with large or complex aggregations, you can automatically parallelize the operation with PLINQ (see Chapter 22).

Unseeded aggregations

You can omit the seed value when calling `Aggregate`, in which case the first element becomes the *implicit* seed, and aggregation proceeds from the second element. Here's the preceding example, *unseeded*:

```
int[] numbers = { 1, 2, 3 };
int sum = numbers.Aggregate ((total, n) => total + n);   // 6
```

This gives the same result as before, but we're actually doing a *different calculation*. Before, we were calculating 0 + 1 + 2 + 3; now we're calculating 1 + 2 + 3. We can better illustrate the difference by multiplying instead of adding:

```
int[] numbers = { 1, 2, 3 };
int x = numbers.Aggregate (0, (prod, n) => prod * n);   // 0*1*2*3 = 0
int y = numbers.Aggregate (   (prod, n) => prod * n);   //   1*2*3 = 6
```

As you'll see in Chapter 22, unseeded aggregations have the advantage of being parallelizable without requiring the use of special overloads. However, there are some traps with unseeded aggregations.

Traps with unseeded aggregations

The unseeded aggregation methods are intended for use with delegates that are *commutative* and *associative*. If used otherwise, the result is either *unintuitive* (with ordinary queries) or *nondeterministic* (in the case that you parallelize the query with PLINQ). For example, consider the following function:

```
(total, n) => total + n * n
```

This is neither commutative nor associative. (For example, 1 + 2 * 2 != 2 + 1 * 1.) Let's see what happens when we use it to sum the square of the numbers 2, 3, and 4:

```
int[] numbers = { 2, 3, 4 };
int sum = numbers.Aggregate ((total, n) => total + n * n);    // 27
```

Instead of calculating

```
2*2 + 3*3 + 4*4    // 29
```

it calculates:

```
2 + 3*3 + 4*4      // 27
```

We can fix this in a number of ways. First, we could include 0 as the first element:

```
int[] numbers = { 0, 2, 3, 4 };
```

Not only is this inelegant, but it will still give incorrect results if parallelized—because PLINQ uses the function's assumed associativity by selecting *multiple* elements as seeds. To illustrate, if we denote our aggregation function as follows:

```
f(total, n) => total + n * n
```

LINQ to Objects would calculate this:

```
f(f(f(0, 2),3),4)
```

whereas PLINQ might do this:

```
f(f(0,2),f(3,4))
```

with the following result:

```
First partition:   a = 0 + 2*2  (= 4)
Second partition:  b = 3 + 4*4  (= 19)
Final result:          a + b*b  (= 365)
OR EVEN:               b + a*a  (= 35)
```

There are two good solutions. The first is to turn this into a seeded aggregation with 0 as the seed. The only complication is that with PLINQ, we'd need to use a special overload in order for the query not to execute sequentially (see "Optimizing PLINQ").

The second solution is to restructure the query such that the aggregation function is commutative and associative:

```
int sum = numbers.Select (n => n * n).Aggregate ((total, n) => total + n);
```

> **NOTE**
>
> Of course, in such simple scenarios you can (and should) use the `Sum` operator instead of `Aggregate`:
>
> ```
> int sum = numbers.Sum (n => n * n);
> ```
>
> You can actually go quite far just with `Sum` and `Average`. For instance, you can use `Average` to calculate a root-mean-square:
>
> ```
> Math.Sqrt (numbers.Average (n => n * n))
> ```
>
> You can even calculate standard deviation:
>
> ```
> double mean = numbers.Average();
> double sdev = Math.Sqrt (numbers.Average (n =>
>               {
>                 double dif = n - mean;
>                 return dif * dif;
>               }));
> ```
>
> Both are safe, efficient, and fully parallelizable. In Chapter 22, we give a practical example of a custom aggregation that can't be reduced to `Sum` or `Average`.

Quantifiers

`IEnumerable<TSource>`→*bool*

Method	Description	SQL equivalents
`Contains`	Returns `true` if the input sequence contains the given element	`WHERE ... IN (...)`
`Any`	Returns `true` if any elements satisfy the given predicate	`WHERE ... IN (...)`
`All`	Returns `true` if all elements satisfy the given predicate	`WHERE (...)`
`SequenceEqual`	Returns `true` if the second sequence has identical elements to the input sequence	

Contains and Any

The `Contains` method accepts an argument of type `TSource`; `Any` accepts an optional *predicate*.

`Contains` returns `true` if the given element is present:

```
bool hasAThree = new int[] { 2, 3, 4 }.Contains (3);       // true;
```

`Any` returns `true` if the given expression is true for at least one element. We can rewrite the preceding query with `Any` as follows:

```
bool hasAThree = new int[] { 2, 3, 4 }.Any (n => n == 3);  // true;
```

`Any` can do everything that `Contains` can do, and more:

```
bool hasABigNumber = new int[] { 2, 3, 4 }.Any (n => n > 10);  // false;
```

Calling `Any` without a predicate returns `true` if the sequence has one or more elements. Here's another way to write the preceding query:

```
bool hasABigNumber = new int[] { 2, 3, 4 }.Where (n => n > 10).Any();
```

`Any` is particularly useful in subqueries and is used often when querying databases; for example:

```
from c in dbContext.Customers
where c.Purchases.Any (p => p.Price > 1000)
select c
```

All and SequenceEqual

`All` returns `true` if all elements satisfy a predicate. The following returns customers whose purchases are less than $100:

```
dbContext.Customers.Where (c => c.Purchases.All (p => p.Price < 100));
```

`SequenceEqual` compares two sequences. To return `true`, each sequence must have identical elements, in the identical order. You can optionally provide an equality comparer; the default is `EqualityComparer<T>.Default`.

Generation Methods

`void→IEnumerable<TResult>`

Method	Description
`Empty`	Creates an empty sequence
`Repeat`	Creates a sequence of repeating elements
`Range`	Creates a sequence of integers

`Empty`, `Repeat`, and `Range` are static (nonextension) methods that manufacture simple local sequences.

Empty

`Empty` manufactures an empty sequence and requires just a type argument:

```
foreach (string s in Enumerable.Empty<string>())
  Console.Write (s);                              // <nothing>
```

In conjunction with the `??` operator, `Empty` does the reverse of `DefaultIfEmpty`. For example, suppose that we have a jagged array of integers and we want to get all the integers into a single flat list. The following `SelectMany` query fails if any of the inner arrays is null:

```
int[][] numbers =
{
  new int[] { 1, 2, 3 },
  new int[] { 4, 5, 6 },
  null                     // this null makes the query below fail.
};

IEnumerable<int> flat = numbers.SelectMany (innerArray => innerArray);
```

`Empty` in conjunction with `??` fixes the problem:

```
IEnumerable<int> flat = numbers
  .SelectMany (innerArray => innerArray ?? Enumerable.Empty <int>());

foreach (int i in flat)
  Console.Write (i + " ");     // 1 2 3 4 5 6
```

Range and Repeat

`Range` accepts a starting index and count (both integers):

```
foreach (int i in Enumerable.Range (5, 3))
  Console.Write (i + " ");                    // 5 6 7
```

`Repeat` accepts an element to repeat, and the number of repetitions:

```
foreach (bool x in Enumerable.Repeat (true, 3))
  Console.Write (x + " ");                    // True True True
```