LINQ

Version .NET 10 · verified 2026-08-25

Language-Integrated Query in C# — the two syntaxes, deferred execution, the everyday operators, and what changes when the source is EF Core.

Contents

Mental model 2

Two syntaxes, one result 2

Filtering, projecting, ordering 3

Deferred execution 4

Aggregates and single elements 5

Grouping and joining 7

LINQ over EF Core 8

Further reading 9

Mental model

LINQ is one query vocabulary over every data source: the same `Where` and `Select` run against a `List<T>` in memory, an XML document, or a SQL table. Those operators are extension methods on `IEnumerable<T>` and `IQueryable<T>`, and they build a query rather than run one — nothing happens until something enumerates it. Which interface you are holding decides where the work happens: in your process, or as SQL on a database server.

Two syntaxes, one result

Query keyword	Method
`from x in xs`	The source; repeated → `SelectMany`
`where`	`Where`
`select`	`Select`
`orderby`, `descending`	`OrderBy`, `ThenBy`
`group … by … into`	`GroupBy`
`join … on … equals`	`Join`, `GroupJoin`

“The compiler translates expressions using these keywords to the equivalent method calls. The two forms are synonymous.” Method syntax is the common choice in .NET codebases because it chains and because many operators — `Count`, `Any`, `Skip`, `ToList` — have no keyword at all. Pick one style per codebase and stay with it.

```
// The same query, spelled two ways.
var q1 = from s in students
         where s.Year == GradeLevel.FirstYear
         orderby s.LastName
         select s.FirstName;

var q2 = students
  .Where(s => s.Year == GradeLevel.FirstYear)
  .OrderBy(s => s.LastName)
  .Select(s => s.FirstName);
```

> **Tip:** Query syntax earns its keep with `let`, which binds an intermediate value once and reuses it later in the query. Method syntax has no equivalent — you either recompute the expression or project an anonymous type to carry it.

Filtering, projecting, ordering

Method	Does
`Where(p)`	Keeps the elements that match
`Select(f)`	Projects each element to a new shape
`SelectMany(f)`	Flattens a sequence of sequences
`OrderBy(k)`, `ThenBy(k)`	Sorts, then breaks ties
`Skip(n)`, `Take(n)`	Pages through a sequence
`Distinct()`, `DistinctBy(k)`	Drops duplicates

This is the bulk of everyday LINQ: filter, reshape, sort, page. `Select` is where DTOs and anonymous types get built, and it is the operator that turns “rows I loaded” into “exactly the fields I need”. `SelectMany` flattens one level — the fix whenever you find yourself nesting a `foreach` inside a `foreach`.

```
var page = orders
  .Where(o => o.Total > 100m)
  .OrderByDescending(o => o.Placed)
  .Skip(20)
  .Take(20)
  .Select(o => new { o.Id, o.Total });

// One flat sequence of every line item.
var lines = orders.SelectMany(o => o.Lines);
```

> **Gotcha:** No LINQ operator changes its source. `orders.OrderBy(…)` on its own sorts nothing — it returns a new sequence, and dropping that return value is the most common first-week LINQ bug.

Deferred execution

Call	When it runs
`Where`, `Select`, `OrderBy`	Deferred, on enumeration
`Sum`, `Count`, `First`, `Any`	Immediately
`ToList()`, `ToArray()`	Immediately, and buffers

Call	When it runs
`ToDictionary()`, `ToLookup()`	Immediately, and buffers

“Deferred execution means that the evaluation of an expression is delayed until its realized value is actually required.” Operators that return a sequence defer; those returning a single value execute on the spot. So a query variable is a recipe, not a result — it reads the source as it is at enumeration time, not at declaration time.

```
var nums = new List<int> { 1, 2, 3 };
var q = nums.Where(n => n > 2); // nothing ran

nums.Add(5);
foreach (var n in q) { }  // runs now: 3, 5
var count = q.Count();    // runs a second time
```

> **Gotcha:** Every enumeration re-executes the whole chain. Two `foreach` loops over one query variable do the work twice — and over EF Core, that is two round-trips to the database. Call `ToList()` once when you need the results more than once.

Aggregates and single elements

Method	Returns
`Any()`, `All(p)`	`bool`; `Any` stops at the first hit

Method	Returns
`Count()`, `Sum()`, `Average()`	A scalar, immediately
`First(p)`, `FirstOrDefault(p)`	Throws, or `default(T)`
`Single(p)`, `SingleOrDefault(p)`	Throws if two match
`CountBy(k)`, `AggregateBy(…)`	Per-key totals (.NET 9+)

Reach for `Any()` over `Count() > 0`: it stops as soon as one element matches, while `Count()` may walk the entire sequence. `Single` is an assertion — use it when a second match means a bug, and `First` when you simply want the first of several. .NET 9 added `CountBy` and `AggregateBy`, which "aggregate state by key without needing to allocate intermediate groupings via `GroupBy`".

```
bool overdue = invoices.Any(i => i.IsOverdue);

// Frequency per key, without GroupBy.
var perStatus = invoices.CountBy(i => i.Status);

// Throws if the number is not unique.
var inv = invoices.Single(i => i.No == "A-1");
```

> **Gotcha:** `FirstOrDefault` on a sequence of value types returns `0`, not `null`, so "found nothing" and "found a real zero" look identical. Query a nullable projection, or test with `Any` first.

Grouping and joining

Method	Produces
`GroupBy(k)`	`IGrouping<K,T>` per key, deferred
`ToLookup(k)`	The same shape, built immediately
`Join(…)`	Inner join on matching keys
`GroupJoin(…)`	Each left item with its matches
`LeftJoin(…)`, `RightJoin(…)`	Outer joins (.NET 10+)

`GroupBy` gives a sequence of groups, each one a key plus its elements, so aggregating per key is a `Select` over the groups. `ToLookup` is its eager twin: it runs immediately and returns an indexable lookup, which is what you want when the same grouping is queried repeatedly.

```
var byStatus = orders
  .GroupBy(o => o.Status)
  .Select(g => new { g.Key, N = g.Count() });


var rows = students.LeftJoin(
  departments,
  s => s.DepartmentId,
  d => d.Id,
  (s, d) => new { s.Name, Dept = d?.Name });
```

> **Tip:** `LeftJoin` and `RightJoin` are new in .NET 10 and EF Core 10 translates them to real SQL outer joins. Before that, a left outer join meant `GroupJoin` plus `SelectMany` plus `DefaultIfEmpty` — if you see that trio in a codebase, this is what it was working around.

LINQ over EF Core

Piece	Meaning
`IEnumerable<T>`	Runs in your process
`IQueryable<T>`	Becomes an expression tree, then SQL
`AsEnumerable()`, `ToList()`	Switches to client evaluation
`ToListAsync()`, `AnyAsync()`	EF Core's async materializers
`AsNoTracking()`	Read-only; skips change tracking

"For `IQueryable<T>`, the query is translated into an expression tree", which EF Core turns into SQL — so `Where` and `Take` become a `WHERE` and a `TOP`, and only the matching rows travel. Project with `Select` to a DTO so the `SELECT` lists just the columns you use, and add `AsNoTracking()` for read-only queries. The async operators live in `Microsoft.EntityFrameworkCore`, not `System.Linq`.

```
var rows = await db.Orders
  .AsNoTracking()
  .Where(o => o.Total > 100m)
  .OrderByDescending(o => o.Placed)
  .Take(20)
```

```
  .Select(o => new OrderDto(o.Id, o.Total))
  .ToListAsync();
```

> **Warning:** `AsEnumerable()` or `ToList()` in the middle of a query moves everything after it into memory — EF Core loads the whole table, then filters. EF Core throws at runtime for untranslatable code outside the top-level projection precisely to stop that happening quietly; a C# helper method inside `Where` is the usual trigger.

Further reading

- LINQ overview
- Standard query operators
- Deferred execution and lazy evaluation
- System.Linq.Enumerable API reference
- EF Core — client vs. server evaluation
- EF Core — efficient querying