LINQsights

Deep-dives, performance analysis and tips against pitfalls

by David Ritter

devritter @github

Query Syntax or Method Syntax?

```
(from x in LinqKnowledgeBase.GetAllInsights()
 where x.IsDeveloperRelevant && !x.IsWellKnown
 orderby x.Fancyness descending
 select new MeetupSlide(x.Headline, x.Content, x.DemoCode))
.Take(meetup.MaxSlides)
.TakeWhile(_ => !timer.IsTimeOver())
.ToList(); // or ToArray()?
```

First some basics!

Before LINQ

- Nested loops
- Queries against data sources with strings
- Different APIs for different data sources

With LINQ

- Query syntax integrated into the C# language => **L**-anguage **IN**-tegrated **Q**-uery
- Same query and transformation pattern against differerent data sources
- Type checking
- Expression Trees
- Lazy Evaluation

Scope of This Presentation

- ✅ IEnumerable and ✅ item streaming
- ✅ performance considerations
- ✅ pitfalls
- ✅ tips and tricks
- ❌ LINQ to Objects / SQL / Entities
- ❌ IQueryable
- ❌ Expression<>
- ❌ PLINQ
- ✅ LINQPad

LINQPad / NetPad (free)

- Super tools for C# scripting!
- no need for `.sln`, `.csproj`
- easily connect to databases and query data the LINQ-way!
- or just write some console apps :)
- call methods from your assemblys
- `.Dump()` takes everything and displays it nicely!
- LINQPad
- NetPad

Technical Basics

- `IEnumerable<T>`
- Extension methods
- Object Initializers `new SomeClass { PropA = x.Name }`
- Anonymouse Types `new { x.Name }`
- Local variable type inference `var`
- Lambda expressions `x => x.Size > 10`
- Expression Trees (for `IQueryable<T>`)

What is an `IEnumerable<T>`?

- An interface
  - with one method: `IEnumerator<T> GetEnumerator()`
    - with mainly `Current` property and `bool MoveNext()` method
- implemented by:
  - `List<T>`
  - `T[]`
  - `Dictionary<TKey, TValue>`
  - `HashSet<T>`
  - `string` => `IEnumerable<char>`

Can I create `IEnumerable<T>`s myself?

Of course!

```
public static IEnumerable<string> MyItemFactory()
{
        yield return "I will";
        yield return "always return";
        yield return "the same starting text";

        if (DateTime.Now.DayOfWeek is DayOfWeek.Saturday or DayOfWeek.Sunday)
        {
                yield return "Did you know it's weekend?";
        }
        if (DateTime.Now.Year == 2012)
        {
                yield return "the end is near!!!";
                yield break;
        }

        yield return "thanks for iterating!";
}
```

What if I don't have `IEnumerable<T>` 😱?

```
var myString = "Hello Meetup people!";
var matches = Regex.Matches(myString, "Meetup");
matches.First(); // <-- compile error!
```

Fixes

- `matches.OfType<Match>()`
- `matches.Cast<Match>()`
- `matches.OfType<object>()` / `matches.Cast<object>()`


I ALWAYS USE LIST<T>
THAT'S OK, RIGHT?
imgflip.com

I use Lists / `.ToList()` always - that's ok, right?

Let's parse a CSV file

- Warehouse and retail sales
- 26 MB
- ~300k Lines

Question 1: How many items of type "WINE" are inside of this file?

Question 2: What's the RAM usage for that application?

I use Lists / `.ToList()` always - that's ok, right?

- depends on what you will do with your data
- if you need to iterate over it multiple times
  - do you always need ALL data?
  - how FAST is your data source? (e.g. disk speed)
  - do some benchmarking!

.ToList() or .ToArray()?

Which is faster?

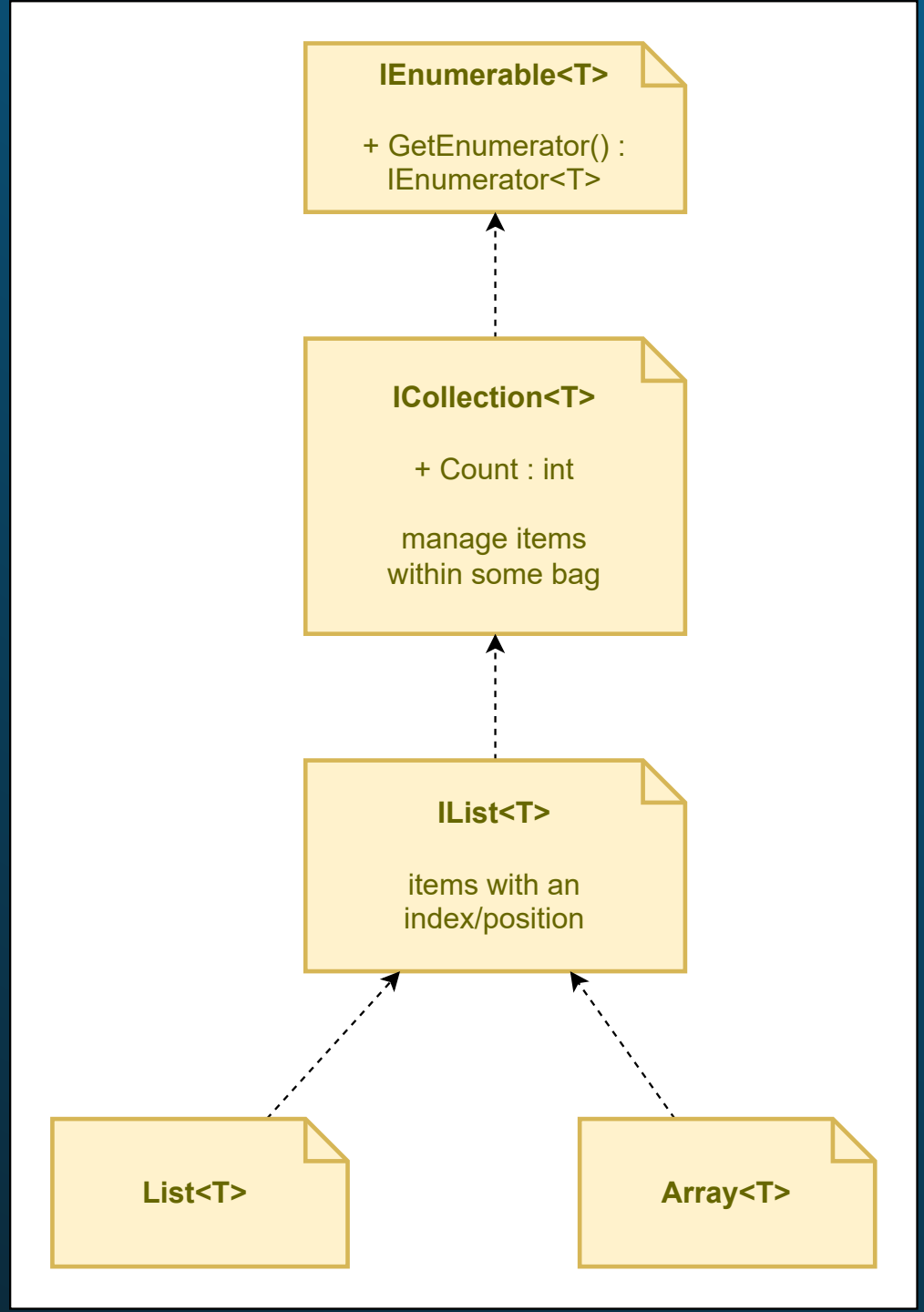
- Check if that's really your performance bottleneck ;)
- If your source is an `ICollection` already, it does not really make a big difference
- Do you really need everything in memory?
- depends on how you will modify the resulting collection

20-tolist-toarray.linq, 21-tolist-toarray-wheretrue.linq

So I make methods for `List<T>` AND `T[]` to be performant?

No, there is a better option!

`ICollection<T>`



Materialize `.GroupBy()`

- Materialization = pull the data into memory
- like `.ToList()` and `.ToArray()`
- `.GroupBy()` still has deferred / lazy execution
- 3 options to materialize it:
  - `.ToList()` => `List<IGrouping<TKey, TValue>>`
  - `.ToDictionary()` => `Dictionary<TKey, List<TValue>>`
  - `.ToLookup()` => `ILookup<TKey, TValue>`
    - readonly!

Don't know if `.Count()` could be expensive?

e.g. for Pagination controls

- use `TryGetNonEnumeratedCount(out int count)`
- even works with some other LINQ methods in place, like `.Reverse()` or `.Take()`

```
.Count() > 0
```

Problems:

- reading all lines / files / ...
- to count every item ...
- just to say "oh yeah, there is something!"

`.Count() > 0` 

"Better" 🙈 (from performance perspective):

- `.Count > 0` if you have an `ICollection<T>` / `List<T>`
- `.Length > 0` if you have an Array

But more expressive:

- `.Any()`
- `.HasContent()` => **BlazingExtensions**

`.Count()` / `.Any()` more readings

- CA1860: Avoid using 'Enumerable.Any()' extension method
- CA1827: Do not use Count()/LongCount() when Any() can be used
- CA1829: Use Length/Count property instead of Enumerable.Count method
- => make a team decision

Checking for empty collections

Difficult to read:

- `myItems.Count() == 0`
- `myItems.Count == 0`
- `!myItems.Any()`
- `!myItems?.Any() ?? true`

Solution:

- `myItems.LacksContent()` => **BlazingExtensions**

Beware of empty lists!

Which problems can occur here?

```
someNumbers.Min();
someNumbers.Max();
someNumbers.Average();
someNumbers.Sum();
```

Beware of empty lists!

Solution:

```
someNumbers = someNumbers.DefaultIfEmpty();
someNumbers.Min();
someNumbers.Max();
someNumbers.Average();
someNumbers.Sum();
```

Tipps for `null` items

- some methods like `.Sum()`, `.Min()`, `.Max()` just ignore `null` numbers
- think about what you want when using `.Count()`!
- to unwrap nullables use `.WhereNotNull()` from **BlazingExtensions**

Everything `All()`-right?

```
int[] someNumbers = [1,5,10];

someNumbers.All(x => x > 0);
// returns true

someNumbers.All(x => x > 10);
// returns false

someNumbers.Where(x => x > 10).All(x => x > 10);
// returns ?
```

- => `.BzAll()` from **BlazingExtensions**

Use `for` -loop features in LINQ

Your tasks:

1. only take every 10th element of a list
2. select ViewModel objects and set their ListIndex property

Solutions:

1. `items.Where((item, index) => index % 3 == 0)`
2. `items.Index().Where(x => new ViewModel { Index = x.Index, Name = x.Item.Name })`

Is LINQ slower than `for` or `foreach`?

- it adds some overhead
- but I would not call it "slow"
- low-level operations (like `for`) are usually faster
- but Assembler would be even faster, so why not use Assembler?
- => Check for the real bottleneck!

One complexe `.Where()` or many small `.Where()`?

- the LINQ overhead adds up
- Behavior with `IQueryable<T>` could be different
- => prefer readable code!
- => make a team decision!
- => Check for the real bottleneck!

Skip `.Where()` when possible

- the predicate function needs to be invoked for every item
- if some conditions are the same for the whole iteration, we can get rid of the `WhereIterator`
- but only measurable for large lists!
- => prefer readable code!

85-skip-where.linq, 86-skip-where-solution.linq

Should I use `.Where(x => ...).Count()` or `.Count(x => ...)`?

- Regarding the small LINQ overhead `Count(x => ...)` should be faster
- When using `IQueryable<T>` you will have the same resulting SQL
- GitHub Issue
-

```
| Method          |      Mean |     Error |    StdDev |   Gen0 | Allocated |
| --------------- | --------: | --------: | --------: | -----: | --------: |
| LinqWhereCount | 1.618 us | 0.0319 us | 0.0391 us | 0.0229 |      72 B |
| LinqCount       | 2.238 us | 0.0257 us | 0.0228 us | 0.0114 |      40 B |
```

- => make a team decision!

Watch out for silly calls!

- `.Where(x => x.SomeFlag).Where(x => x.SomeFlag)`
- `.Distinct().Distinct()`
  - Distinct is quite expensive
- `Distinct().Count() > 0`
  - Why clean up list just to check if there is something?
- `.OrderBy(x => x.Name).ToHashSet()`
  - A `HashSet<T>` does not care about ordering
- `.OrderBy(x => x.Name).Count()`
  - why sort the work before counting it?

List flattening with `.SelectMany()`

```
var peopleWithPets = new[] {
    new { Name = "Anna", Pets = new[] { "Dog", "Cat", "Parrot" } },
    new { Name = "Ben",  Pets = new[] { "Fish", "Hamster", "Snake" } }
};

var allPets = peopleWithPets.SelectMany(p => p.Pets);

foreach (var pet in allPets)
{
    Console.WriteLine(pet);
}
```

Write your own extension methods!

DRY => Don't Repeat Yourself!

```
public interface ISoftDelete
{
    bool IsDeleted { get; set; }
}

public static class MyExtensions
{
    public static IEnumerable<T> WhereNotDeleted<T>(this IEnumerable<T> source)
        where T : ISoftDelete
    {
        return source.Where(x => x.IsDeleted == false)
    }
}
```

Write your own extension methods!

- `.WhereActive()` with `IIsActive` items
- `.WhereNotExpired()`
- `.ApplyDefaultSorting()`
- `.ApplySearchFilter(ProductSearchFilter filter)`
- `.Paginate(PaginationQuery query)`
- `.Paginate(source, pageNumber, pageSize)`
- `.GroupByCategory()`
- `.CountNulls()`
- `.Shuffle()`

How to reuse LINQ parameters

```
public static class MyFilters
{
    public static Func<User, bool> IsActive = u => u.IsActive;
    public static Func<User, bool> IsAdult = u => u.Age >= 18;
    public static Func<User, bool> IsActiveAndAdult = u => IsActive(u) && IsAdult(u);
}

var result1 = users.Where(MyFilters.IsActive).Where(MyFilters.IsAdult);
var result2 = users.Where(MyFilters.IsActiveAndAdult);
```

How to reuse LINQ parameters

Further ideas:

- `UserSorter`
- `ProductByCategoryThenNameSorter`
- `UserComparer`
- `ProductCountSelector`
- `TotalPriceSelector`
- `AnyActiveNotificationSelector`

Utility Methods

for playing around or unit tests

```
// empty collections
Enumerable.Empty<T>();
Array.Empty<T>();

// create lots of numbers
Enumerable.Range(1_000_000, 2_000_000);

// create a huge list for processing
Enumerable.Repeat(new ComplexObject(), 1_000_000);)
```

Thanks for listening!

Further links:

- BlazingExtensions on GitHub and NuGet
- MoreLINQ
- Hyperlinq
- StructLinq
- BenchmarkDotNet
- LINQPad
- NetPad