EF CORE MIT MICROSOFT SQL SERVER

Performance-Falle

Asynchrone Abfragen großer Datenmengen beschleunigen.

Das Entity Framework hat sich in der .NET-Welt für die Arbeit mit relationalen Datenbanken etabliert. Provider für EF Core gibt es mittlerweile für alle gängigen Datenbanksysteme. In der .NET-Welt besonders beliebt ist Microsofts hauseigene Datenbank, der SQL Server (MSSQL). Auch für asynchrone Abfragen mittels *async* und *await* hat sich das Entity Framework bewährt. Allerdings gibt es gerade bei der beliebten Konstellation von EF Core und Microsoft SQL in Kombination mit *async* und *await* immer wieder Performance-Probleme. Deutlich machen soll dies folgendes Minibeispiel:

```
public class ProblemDbContext : DbContext
{
  public DbSet<ProblemTable> Problem { get; set; }
  protected override void OnConfiguring(
      DBContextOptionsBuilder optionsBuilder)
  {
    optionsBuilder.UseSqlServer(connectionString:
      @Server=(localdb)\MSSQLLocalDb;
      Initial Catalog=ProblemDb;)
  }
}

public class ProblemTable
{
  public Guid ID { get; set; }
  public string LargeString { get; set; }
  public byte[] LargeBinary { get; set; }
}
```

Es wird dafür ein einfacher *DbContext* mit einer einzigen Tabelle erzeugt. Das Performance-Problem, um das es hier geht, macht sich insbesondere bei Tabellen bemerkbar, die umfangreiche Binärdaten enthalten. Im Minibeispiel ist das die *ProblemTable*, die neben der Zeichenkette *LargeString* auch noch ein großes Byte-Feld *byte[] LargeBinary* enthält.

Es folgen ein paar Beispieldaten, mit denen die Tabelle gefüllt wird: Es sind zehn Einträge, jeweils mit einem zufällig mit 10 MByte gefüllten Byte-Feld plus einem 1024 Zeichen langen String. Es geht also nicht um atemberaubend unhandliche Datenmengen:

```
await using var ctx = new ProblemDbContext();
ctx.Database.EnsureCreated();
var buffer = new byte[1024 * 1024 * 10]
Random.Shared.NextBytes(buffer);
ctx.Problem.AddRange(Enumerable.Range(0, 10)
    .Select(x:int => new ProblemTable()
{
  LargeBinary = buffer,
  LargeString = new string(c: 't', count: 1024)
}));
await ctx.SaveChangesAsync();
```

Bei der Anlage des Beispiels gibt es mit diesem Code noch kein Problem. Richtig schmerzhaft wird allerdings die Abfrage von Daten aus diesem Minibeispiel.

Jetzt wird es langsam

Der „schmerzhafte" Code gliedert sich in drei Teilabschnitte: In Phase 1 wird eine neue Instanz gebaut und der Kontext mit *await* abgefragt. Damit ist der Kontext bereit, das Modell von EF Core wurde erzeugt und EF Core selbst ist „warmgelaufen". Phase 2 ruft danach einfach den ersten Datensatz ab, und zwar asynchron (*FirstOrDefaultAsync*), wohingegen Phase 3 den gleichen Datensatz synchron abruft (*FirstOrDefault*).

```
// Phase 1
await using var initialContextToWarmUp =
  new ProblemDbContext();
await initialContextToWarmUp.Problem
  .FirstOrDefaultAsync();

// Phase 2
var asyncWatch = Stopwatch.StartNew();
await using var asyncContext = new ProblemDbContext();
await asyncContext.Problem.FirstOrDefaultAsync();
Console.WritLine(asynWatch.Elapsed);

// Phase 3
var synWatch = Stopwatch.StartNew();
await using var syncContext = new ProblemDbContext();
synContext.Problem.FirstOrDefault();
Console.Writline(syncWatch.Elapsed);
```

Man könnte meinen, dass beide Abfragevarianten gleich schnell oder zumindest ähnlich schnell sein sollten. Bild 1 zeigt die Laufzeiten in der traurigen Realität: Die asynchrone Abfrage dauert rund 40-mal so lang (6,3101 Sek.) wie die syn-

Katastrophale Werte für die asynchrone Abfrage (oben) – unten die Werte der synchronen Abfrage (Bild 1)

```
00:00:06.3101399
00:00:00.1549630
```

Method	Mean	Error	StdDev	Gen 0	Gen 1	Gen 2	Allocated
Async	3,170.53 ms	222.636 ms	132.487 ms	635000.0000	634000.0000	633000.0000	13155.71 MB
Sync	37.88 ms	2.139 ms	1.415 ms	333.3333	333.3333	333.3333	20 MB

Die Community berichtet: Es wird schlimmer, je genauer man hinsieht (Bild 2)

chrone Abfrage (0,1549 Sek.). Das ist nicht nur „ein bisschen mehr", sondern ein sagenhaft großer Laufzeitunterschied.

Woran liegt das?

Microsoft ist dieser Umstand längst bekannt. Schon seit Juni 2020 gibt es auf GitHub einen Problembericht (Issue) [1], der mittlerweile seit fast fünf Jahren auf einen Fix wartet.

Das EF-Core-Team sieht sich dabei als nicht zuständig an, weil der eigentliche Fehler im Treiber von Microsofts SQL Server liegt. Der Problembericht von EF Core referenziert nämlich ein Issue im Treiber, das ebenfalls seit Juni 2020 auf eine Lösung wartet [2]. Dort wurde das Problem bereits tiefergreifend analysiert, wie zum Beispiel der kanadische Entwickler Davoud Eshtehari unter [2] schreibt.

In Bild 2 sehen Sie einige Ergebnisse einer Messung mit BenchmarkDotNet, die nicht nur belegen, dass die Laufzeit unterirdisch schlecht ist, sondern auch, dass zugleich der Speicherbedarf atemberaubend groß ausfällt.

Mit anderen Worten: Alle Entwickler, die EF Core und Microsoft SQL Server in Verbindung mit *async* und *await* nutzen, haben dasselbe Problem bei allen Tabellen, die große Binärdaten enthalten. Und seien wir ehrlich: Das sind einige Projekte.

Kein Fix in Sicht?

Microsoft ist sich des Problems zwar offenbar bewusst, aber die Mühlen in Seattle mahlen recht langsam. Immerhin gibt es bereits eine „Teillösung". Dafür verlangt EF Core mindestens die Version 5.1.6 des *Microsoft.Data.SqlClient*-Treibers. Besser wechselt man gleich zur neuesten Treiberversion 6.0.1, die vor rund zwei Monaten veröffentlicht wurde. Nutzt man diese, so ändert sich die Laufzeit des Mini-Beispiels ohne eine weitere Anpassung dramatisch. Anstatt mehr als sechs Sekunden benötigt der asynchrone Code jetzt nur noch rund 1,5 Sekunden.

Noch etwas mehr optimieren?

Es gibt allerdings noch eine weitere Optimierung, um den asynchronen Weg schneller zu machen. Das Grundproblem, das im Microsoft-Treiber steckt, ist offenbar die Paketverarbeitung des SQL-Server-Protokolls. Und hier gibt es eine Stellschraube für Optimierungen. Dafür gibt man dem *ConnectionString* über den Parameter *Packet Size* mit, dass man gern größere, dafür aber weniger Pakete erhalten möchte:

```
optionsBuilder.UseSqlServer(connectionString:
  @Server=(localdb)\MSSQLLocalDb;
  Packet Size=32766;
  Initial Catalog=ProblemDb;)
```

Die asynchrone Abfrage benötigt 0,33 Sekunden und dauert damit dreimal so lang wie die synchrone Variante (Bild 3)

```
00:00:00.3348248
00:00:00.1103132
```

Mit dieser Vorgabe liefert die Datenbank das Ergebnis noch einmal ein gutes Stück schneller als zuvor. Anstelle der ursprünglichen sechs Sekunden benötigt die asynchrone Abfrage lediglich 0,33 Sekunden und ist damit nur noch um den Faktor 3 langsamer als die synchrone Variante (Bild 3).

Fazit

Geliebtes Microsoft, so gern ich das .NET-Ökosystem auch habe: Ein solcher Fehler darf doch nicht passieren! Vor allem die lange Dauer bis zum Fix des Problems ist enttäuschend. Zwar kommuniziert Microsoft dies transparent in einem Video vom November 2024 [3], aber das ist dann doch zu wenig. Schließlich leidet die Performance, indem man Microsoft-Technologien und -Produkte einsetzt. Das Problem besteht nämlich nur, wenn man mit dem SQL Server arbeitet; Entwickler, die zum Beispiel auf PostgreSQL setzen, kennen das geschilderte Performance-Problem nicht, denn der PostgreSQL-Treiber arbeitet genau so, wie er es soll.

Wer also die Kombination aus SQL Server, EF Core und asynchronen Abfragen nutzen muss, sollte die vorgestellten Tipps berücksichtigen, mit denen sich asynchrone Abfragen deutlich beschleunigen lassen – auch wenn sie immer noch spürbar langsamer als synchrone Abfragen sind. ■

[1] GitHub, Issue #21147 zu EF Core, www.dotnetpro.de/SL2504-05NETirol1
[2] GitHub, Reading large data (binary, text) asynchronously is extremely slow, www.dotnetpro.de/SL2504-05NETirol2
[3] Video: Navigating Async Challenges in EF Core, www.dotnetpro.de/SL2504-05NETirol3

Christian Giesswein
studierte Wirtschaftsinformatik in Wien und entwickelt von klein auf Software mit .NET und C#. In Tirol hat er das Unternehmen Giesswein Software-Solutions gegründet, das sich auf Individualsoftware und Consulting spezialisiert hat.
christian@software.tirol

dnpCode A2504-05NETirol