En kvantitativ studie av prestanda hos olika formulerade LINQ-uttryck för logiskt likvärdiga predikat i Entity Framework Core

Examensarbete 20p

Författare: Emil Uppenberg

Program/Kurs: Systemutveckling .NET

Handledare: Marcus Friberg

Datum: 19 Maj 2026

Abstract

Entity Framework Core (EF Core) is a programming framework that simplifies communication with relational database systems in .NET development. A core principle of EF Core is that read and write operations against the database are performed by writing LINQ expressions, which are translated into SQL queries in the SQL dialect of the selected database system. This simplifies the code that handles database communication, but the abstraction also hides how the database performs the operations. Different LINQ expressions can produce the same logical result from a C# perspective, while being translated into different SQL queries with different execution plans and response times.

This study examines different implementations of LINQ in a context where communication takes place with a SQL Server database through EF Core, in order to draw conclusions from a performance perspective. To measure different LINQ expressions from the client's perspective, an application was built with BenchmarkDotNet in which the LINQ expressions are tested against a SQL Server database containing pre-generated data. The generated SQL queries were also executed in SQL Server Management Studio to examine the execution plans of the LINQ expressions more closely and compare the database's execution time with the client's.

The results show that when verifying the existence of objects in object collections, Any() leads to significantly faster execution time than Count() > 0 when the retrieved data is not materialized, because SQL Server can return a result at the first match. Normalizing strings inside LINQ expressions prevents SQL Server from using indexes to create efficient execution plans. Projecting and materializing a selection of columns from one or more tables leads to significantly faster execution time and lower memory usage compared with materializing complete entities. For SQL queries involving joins between multiple tables, using AsSplitQuery() leads to significantly faster execution time, especially when several tables are joined at the same level in the object graph. However AsSplitQuery() sends the SQL queries separately which increases traffic load on the database server and may also result in inconsistent data being sent to the client.

Sammanfattning

Entity Framework Core (EF Core) är ett programmeringsramverk som förenklar kommunikation med relationella databas system inom .NET utveckling. En grundpelare i EF Core är att läs- och skrivoperationer mot databasen sker genom programmering av LINQ uttryck som översätts till SQL frågor i det valda databas systemets SQL dialekt. Detta förenklar den kod som hanterar kommunikation med databasen men abstraktionen döljer även hur databasen genomför operationerna. Olika LINQ uttryck kan leda till samma logiska utfall ur ett C# perspektiv men översättas till olika SQL frågor med olika exekveringsplaner och svarstider.

Denna undersökning behandlar olika implementationer av LINQ i ett sammanhang där kommunikation sker mot en SQL Server databas genom EF Core för att kunna dra slutsatser ur ett prestandaperspektiv. För att genomföra mätningar av olika LINQ uttryck ifrån klientens perspektiv har en applikation byggts med BenchmarkDotNet där LINQ uttrycken testas mot en SQL Server databas innehållande förgenererad data. De genererade SQL frågorna har även körts i SQL Server Management Studio för att närmare undersöka exekveringsplanerna för LINQ uttrycken och jämföra databasens arbetstid mot klientens.

Resultaten visar att för att verifiera existens av objekt i objekt samlingar leder Any() till betydligt snabbare exekveringstid än Count() > 0 när den lästa datan inte materialiseras, eftersom SQL Server då kan svara vid första matchning. Normalisering av strängar inom LINQ uttryck medför att SQL Server inte kan använda Index för att skapa effektiva exekveringsplaner. Att projicera och materialisera ett urval av kolumner från en eller flera tabeller leder till betydligt snabbare exekveringstid och mindre minnesanvändning jämfört med att materialisera fullständiga entiteter. För SQL frågor som innebär joins mellan flera tabeller leder användning av AsSplitQuery() till betydligt snabbare exekveringstid, speciellt när flera tabeller joinas på samma nivå i objektgrafen. AsSplitQuery medför dock att SQL frågorna skickas separat och ökar trafikbelastningen på databasservern och kan även leda till att datan som når klienten är inkonsekvent.

Innehållsförteckning

Abstract **2**
Sammanfattning **3**
Innehållsförteckning **4**
Inledning **5**
Syfte och problemställning **6**
Källor **7**
ORM och Entity Framework Core 7
Materialisering och change tracking i Entity Framework Core 7
LINQ 7
Index och SARGable 7
Relationsdatabaser och RDBMS 7
SQL Server, databasstatistik och exekveringsplaner 8
Benchmarking och tolkning av mätvärden 8
Cartesian explosion 9
Metod och genomförande **10**
BenchmarkDotNet 10
SQL Server Management Studio 10
Data 10
Testfall 11
Resultat, analys **14**
Hur bör existens av objekt i en samling verifieras? 14
Hur bör strängar jämföras? 16
Hur bör ett urval av fält från en eller flera tabeller materialiseras? 16
Hur kan hämtning av entiteter med komplexa relationer förbättras? 18
Slutsats och diskussion **20**
Hur bör existens av objekt i en samling verifieras? 20
Hur bör strängar jämföras? 20
Hur bör ett urval av fält från en eller flera tabeller materialiseras? 21
Hur kan hämtning av entiteter med komplexa relationer förbättras? 21
Referenser **22**

Inledning

I verkliga applikationer är ett smidigt flöde mellan databasen och klienten avgörande för pålitlighet i affärskritiska processer och även användarens upplevelse. Inom .NET utveckling är EF Core ett kraftfullt verktyg som abstraherar kommunikationen med en databas för att ta fram sådana applikationer. Men abstraktionen medför även att tekniska aspekter döljs bakom kulisserna.

EF Core förenklar läs- och skrivoperationer mot databasen genom att översätta LINQ uttryck till SQL frågor i databasens SQL dialekt. Därför är det viktigt att implementera LINQ på ett sätt som är rimligt för databasen och klienten att optimera utifrån varje falls egna specifika krav.

Denna undersökning tittar närmare på varför olika formulerade LINQ uttryck i logiskt likvärdiga fall leder till olika exekveringstider för att kunna dra slutsatser om fördelar och nackdelar med olika tillvägagångssätt.

Syfte och problemställning

Syftet är att undersöka varför olika LINQ uttryck för ett antal vanligt förekommande logiska predikat kan leda till olika svarstider.

Undersökningen mäter exekveringstid i klienten och ska ge förståelse för hela processen från det att klienten skickar en fråga till databasen genom ett LINQ uttryck tills det att klienten ger respons till användaren.

Undersökningen tittar på hur klientapplikationen kommunicerar med databasen utifrån följande frågeställningar.

- Hur ser fördelningen av arbete ut mellan klienten och servern?
- Vilka faktorer påverkar serverns hantering av frågan?

Undersökningen är uppbyggd runt ett antal testfall och ska leda till slutsatser som kan användas för att förbättra prestanda i liknande fall.

Källor

ORM och Entity Framework Core

Object-Relational Mapping (ORM) är en programmeringsteknik för att abstrahera kopplingen mellan en databas och ett objekt orienterat programmeringsspråk. Entity Framework Core (EF Core) är en ORM som används för att utveckla applikationer med koppling mot en databas i C#.

Materialisering och change tracking i EF Core

När EF Core materialiserar data hämtad ur en databas skapas en objektgraf som sparas i datorns minne. EF Core kan spåra materialiserade objekt och upptäcker då automatiskt ifall objektet modifieras. Detta kallas change tracking och gör det enkelt att spara ändringarna i motsvarande rad i databasen.

LINQ

Language Integrated Query Language (LINQ) är inbyggt i C# och används för att hantera data från itererbara datakällor. LINQ stödjer fluentsyntax som har använts i denna undersökning. Med fluentsyntax är uttryck formulerade som kedjade metoder.

Index och SARGable

Index hjälper SQL Server att hitta matchande rader snabbare. En SQL fråga beskrivs som SARGable (search argument able) när Index kan användas för att snabbt hitta matchande rader.

Relationsdatabaser och RDBMS

En relations databas lagrar data i tabeller. Varje tabell representerar en typ med ett bestämt antal kolumner för olika attribut hos den typen. Varje rad i tabellen är en instans av den typen. Kolumner i en tabell kan vara en nyckel och relationer mellan rader i olika tabeller definieras genom att koppla radernas nycklar till varandra. Relations databas system (RDBMS) är mjukvara som bygger på relations databas modellen.

SQL Server, databasstatistik och exekveringsplaner

SQL Server är ett relations databas system utvecklat av Microsoft. Exekveringsplaner beskriver hur SQL Server fysiskt går tillväga för att svara på en fråga och innehåller en samling steg som leder fram till svaret. I denna undersökning framkommer följande.

- Hash Match är när SQL Server hittar matchande rader genom att bygga ett hash table från en datamängd.
- Bitmap Filter är en teknik för att utesluta garanterade icke-matchande rader för att skynda på processer när tabeller ska joinas i parallela trådar.
- Index Scan är när SQL Server läser hela eller stora delar av en Index för att hitta matchningar.
- Index Seek är när SQL Server kan använda sorteringen på en Index för att direkt söka efter matchningar.

Databasstatistik i SQL Server innehåller information om värdefördelningen i kolumner och Index. Statistiken används för att uppskatta hur många rader en fråga förväntas returnera vilket spelar en roll för vilka strategi SQL Server väljer i exekveringsplaner.

Benchmarking och tolkning av mätvärden

BenchmarkDotNet är ett NuGet paket som mäter prestanda av kod i C# och levererar pålitliga resultat genom följande metoder.

- Uppvärmningskörningar innan faktisk mätning optimerar JIT-kompilatorn, fyller cacheminnen och stabiliserar runtime-miljön.
- Hänsyn tas till mätningarnas overhead.
- Mätningarna körs många gånger och resultatet kan därför baseras på flera mätvärden.

ShowplanXML i SQL Server Management Studio visar värden för följande.

- CpuTime anger summan av tid för SQL Server att exekvera frågan.
- DegreeOfParallelism är antal processorkärnor som används i exekveringsplanen. Om DegreeOfParallelism är högre än 1 är CpuTime summan av tiden samtliga processorkärnor har arbetat.

Cartesian Explosion

Cartesian explosion uppstår när flera tabeller joinas på samma nivå och producerar en ny rad med samtliga kolumner ur båda tabeller för varje relationsobjekt. Detta leder till dataduplicering och högre resursförbrukning.

Metod och genomförande

BenchmarkDotNet

Eftersom studien behandlar mätbara faktorer lämpar sig en kvantitativ metod väl och för att mäta exekveringstid och minnesanvändning har en konsol applikation byggts upp med BenchmarkDotNet. Detta ger konkreta resultat och möjliggör för att kunna resonera utifrån olika tänkbara scenarion och användningsområden.

SQL Server Management Studio

SQL Server Management Studio 22 har använts för att undersöka information om exekveringsplaner för SQL frågor.

Data

Applikationen genererar CSV-filer med testdata genom ett kommando. Testdatan har använts för att uppdatera databasen genom EF Core. Se bilaga 1 för C# koden som tabellerna är skapade utifrån. Relationerna mellan klasserna kan beskrivas enligt följande.

En Customer representerar en kund och kan ha noll eller flera Order kopplade till sig.

En Order representerar en beställning och tillhör en Customer. En Order kan innehålla noll eller flera varubeställningar, representerade av OrderItem. Varje OrderItem tillhör en Order och en Product. En Order kan innehålla noll eller flera leveranshändelser, representerade av ShipmentEvent. Varje ShipmentEvent tillhör en Order.

En Product representerar en produkt och kan ha noll eller flera OrderItem kopplade till sig.

Relationer är distribuerade mellan entiteterna enligt följande regler.

1% av Customers har 0 Orders
1% av Customers har 201-249 Orders
3% av Customers har 26-200 Orders

15% av Customers har 4-25 Orders

80% av Customers har 1-3 Orders

1% av Products har 0 OrderItems

4% av Products har 3392-3393 OrderItems

15% av Products har 404-405 OrderItems

80% av Products har 15-16 OrderItems

1% av Orders har inga ShipmentEvents

Datamängden består av

100,000 Customers

25,000 Products

1,000,000 Orders

6,000,000 OrderItems

3,000,000 ShipmentEvents

Testfall

Hur bör existens av objekt i en samling verifieras?

CountWithHotProductId

```
return db.Customers.Where(x =>
       x.Orders.Count(y =>
              y.Items.Count(z =>
                     z.ProductId == Constants.HotProductId) > 0) > 0)
       .Count() > 0;
```

AnyWithHotProductId

```
return db.Customers.Where(x =>
       x.Orders.Any(y =>
              y.Items.Any(z =>
                     z.ProductId == Constants.HotProductId)))
       .Any();
```

CountWithoutShipmentEvents

```
return db.Customers.Where(x =>
       x.Orders.Count() > 0 &&
       x.Orders.Count(y =>
              y.ShipmentEvents.Count() == 0) > 0)
       .Count() > 0;
```

AnyWithoutShipmentEvents

```
return db.Customers.Where(x =>
       x.Orders.Any() &&
```

```
            x.Orders.Any(y =>
                  !y.ShipmentEvents.Any()))
            .Any();
```

Eftersom att det i praktiken ofta förekommer att datan behöver materialiseras har mätningar gjorts där varje LINQ uttryck materialiseras genom `.ToList()`

Hur bör strängar jämföras?

CompareDefault

```
    return db.Products.Where(x =>
           x.Sku == Constants.HotProductSku)
           .FirstOrDefault() ?? throw new Exception();
```

CompareToLower

```
    return db.Products.Where(x =>
           x.Sku == Constants.HotProductSku.ToLower())
           .FirstOrDefault() ?? throw new Exception();
```

Hur bör ett urval av fält från en eller flera tabeller materialiseras?

Projection

```
    return db.Customers
           .Take(Constants.MaximumCustomers)
           .Select(x => new CustomerProjectionDto(
                  x.Email,
                  x.CountryCode,
                  x.CreatedAtUtc,
                  x.DisplayName));
           .ToList();
```

WholeEntities

```
    return db.Customers
           .Take(Constants.MaximumCustomers);
           .ToList();
```

ProjectionNested

```
    return db.Customers
           .Take(Constants.MaximumCustomers)
           .Select(x => new CustomerProjecttionNestedDto(
                  x.Email,
                  x.CountryCode,
                  x.CreatedAtUtc,
                  x.DisplayName,
                  x.Orders.Select(y => new OrderProjectionNestedDto(
                         y.PaidAtUtc,
                         y.ShippingTotal,
                         y.Subtotal,
                         y.OrderedAtUtc,
                         y.Items.Select(z => new
OrderItemProjectionNestedDto(
                     z.DiscountAmount,
                     z.TaxAmount,
```

```
                    z.Quantity))))))
        .ToList();
```

IncludeEntities

```
return db.Customers
        .Take(Constants.MaximumCustomers)
        .Include(x => x.Orders)
                .ThenInclude(y => y.Items);
        .ToList();
```

Hur kan hämtning av entiteter med komplexa relationer förbättras?

ProjectionNestedSplitQuery

```
return LinqQueries.ProjectionNested(db)
        .AsSplitQuery()
        .ToList();
```

IncludeEntitiesSplitQuery

```
return LinqQueries.IncludeEntities(db)
        .AsSplitQuery()
        .ToList();
```

Resultat, analys

Hur bör existens av objekt i en samling verifieras?

EF Core tittar först på de inre predikaten i LINQ uttryck och därför leder Count() till onödiga summeringar vid jämförelse mot 0. I de genererade SQL frågorna för Count-metoderna måste SQL Server gå igenom hela Customers tabellen och räkna samtliga matchande rader för varje WHERE påstående innan antal Customers som uppfyller kraven kan summeras.

När Any() används är kraven uppfyllda så fort en Customer är hittad med en matchande rad för varje WHERE påstående.

Logiken är fundamentalt olika därför att Count() returnerar antal matchningar på predikatet medans Any() returnerar om en matchning finns eller inte.

Tabell 1. BenchmarkDotNet resultat för Count- och Any-frågor.

Metod	**Medelvärde exekveringstid**	**Antal matchande Customers**
CountWithHotProductId	16 ms	3172
AnyWithHotProductId	3.9 ms	3172
CountWithoutShipmentEvents	209.9 ms	6932
AnyWithoutShipmentEvents	4 ms	6932

Se bilaga 2, sida 1-4, för de generade SQL frågorna.

CountWithHotProductId

Trots att CountWithHotProductId är SARGable genom IX_OrderItems_ProductId blir exekveringsplanen ändå onödigt komplex på grund av SQL-frågans formulering.

SQL Server hittar först 3,392 matchande OrderItems för ProductId = 24,800 via Index Seek på IX_OrderItems_ProductId och använder deras OrderId som bitmapfilter mot en scan av IX_Orders_CustomerId. Eftersom samma OrderId kan förekomma flera gånger dedupliceras resultatet med en Hash Match. Därefter krävs ytterligare steg för att koppla dessa CustomerId till faktiska Customers innan de kan räknas.

AnyWithHotProductId

Exekveringsplanen för AnyWithHotProductId undviker all onödig komplexitet. För att svara på SQL frågan behöver den endast läsa 2 rader från IX_OrderItems_ProductId, 2 rader från PK_Orders, och 1 rad från PK_Customers.

CountWithoutShipmentEvents

CountWithoutShipmentEvents exekveringstid är ungefär tiodubbelt långsammare än CountWithHotProductId därför att SQL frågan saknar ett filtrerande värdepredikat vilket tvingar SQL Server att genomföra Index Scan på IX_ShipmentEvents_OrderId och IX_Orders_CustomerId där samtliga 3,000,000 respektive 1,000,000 rader läses.

AnyWithoutShipmentEvents

Exekveringsplanen för AnyWithoutShipmentEvents är mer effektiv. Eftersom PK_Orders och IX_ShipmentEvents_OrderId båda sorteras i stigande ordning på OrderId behöver SQL Server endast läsa 147 rader ur PK_Orders samt 455 rader ur IX_ShipmentEvents_OrderId innan en OrderId utan ShipmentEvent är funnet.

Tabell 2. BenchmarkDotNet resultat vid materialisering av Count- och Any-resultat.

Metod	**Medelvärde exekveringstid**	**Antal objekt**
CountWithHotProductId_Materialize	35.5 ms	3172
AnyWithHotProductId_Materialize	33.9 ms	3172
CountWithoutShipmentEvent_Materialize	216.5 ms	6932
AnyWithoutShipmentEvents_Materialize	213.8 ms	6932

Se bilaga 2, sida 5-8, för de genererade SQL frågorna.

Materialisering av entiteter

Om datan ska materialiseras leder Any() inte alltid till en fördel eftersom SQL Server då behöver returnera samtliga Customers som uppfyller SQL frågans logiska krav. Exekveringsplanerna som materialiserar Customers är i princip identiska mellan Count- och Any- metoder.

Hur bör strängar jämföras?

Här genererar EF Core nästan identiska SQL frågor där den enda skillnaden är att CompareToLower normaliserar fältet som ska jämföras till lower-case, och kostnaden i exekveringstid för en sådan normalisering är i stort sett obetydlig och förklarar inte skillnaden i exekveringstid mellan metoderna.

Tabell 3. BenchmarkDotNet resultat för strängjämförelser.

Metod	**Medelvärde exekveringstid**
CompareDefault	3.2 ms
CompareToLower	6.7 ms

Se bilaga 2, sida 9-10, för de genererade SQL frågorna.

CompareDefault

SQL frågan för CompareDefault är SARGable genom IX_Products_Sku och därför kan exekveringsplanen söka efter “SKU-00024000” med en Index Seek på IX_Products_Sku och hitta rätt Product direkt.

CompareToLower

SQL-frågan för CompareToLower är inte SARGable eftersom funktionen LOWER appliceras på den indexerade kolumnen Sku. IX_Products_Sku är sorterat efter det lagrade Sku värdet, inte efter uttrycket LOWER(Sku), så SQL Server kan inte göra en Index Seek mot indexet. I stället måste exekveringsplanen scanna IX_Products_Sku och utvärdera LOWER(Sku) rad för rad.

Hur bör ett urval av fält från en eller flera tabeller materialiseras?

Den stora skillnaden mellan SQL frågorna genererade av EF Core för dessa metoder är antalet kolumner som hämtas för varje rad.

Tabell 4. Resultat vid projektion och hämtning av entiteter.

Metod	Medelvärde exekveringstid	Allokerat minne	SQL Server CpuTime
Projection	17.8 ms	4.3MB	8 ms
WholeEntities	57.5 ms	15.6 MB	13 ms
ProjectionNested	1.9 s	388.3 MB	9.3 s / 1.2 s
IncludeEntities	6.2 s	1249 MB	1.7 s

Se bilaga 2, sida 11-14, för de genererade SQL frågorna.

Analys av projektion och hämtning av entitet

Förutom skillnaden i kolumner som returneras så är exekveringsplanerna för Projection och WholeEntities identiska och består endast av en Clustered Index Scan på PK_Customers följt av en Top för att hämta exakt 10,000 rader.

Skillnaden i exekveringstid beror till stor del på olika mängder av data som överförs och materialiseras. Det har även betydelse att objekten som materialiseras i metoderna Projection och ProjectionNested inte innehåller någon entitet-typ och därför inte heller plockas upp av EF Core's change tracker. Detta bidrar till att metoderna får snabbare exekveringstid i BenchmarkDotNet mätningarna.

IncludeEntities

För IncludeEntities använder SQL Server en exekveringsplan som genomför selektiva sökningar med Index Seek. Först läses 10,000 Customers från PK_Customers. Därefter används deras Id för att söka i IX_Orders_CustomerId, vilket resulterar i 87,554 matchande OrderId. För varje OrderId görs sedan sökningar i PK_OrderItems, vilket resulterar i 525,549 matchande OrderItems.

ProjectionNested

För ProjectionNested är valet av exekveringsplan känsligt för vilken andel samplad statistik som finns tillgänglig för de tabeller och Index som berörs.

Med en lägre procent samplad statistik estimeras betydligt fler rader att returneras och SQL Server väljer då samma strategi för exekveringsplanen som IncludeEntities.

Vid högre procent samplad statistik valde SQL Server istället en mindre effektiv exekveringsplan där parallela Index Scan läser alla rader från PK_Customers, PK_Orders och PK_OrderItems och sammanställer dessa genom Hash Match.

Tabell 5. ProjectionNested och IncludeEntities vid olika statistik- och parallelliseringsförhållanden.

Metod	SQL Server CpuTime	Medelvärde exekveringstid	Procent samplad statistik	DegreeOf Parallelism	Allokerat minne
ProjectionNested	9.3 s	3.1 s	Hög	12	388.3 MB
ProjectionNested	1.2 s	1.9 s	Låg	1	388.3 MB
IncludeEntitites	1.7 s	6.2 s	-	1	1249 MB

I SQL Server är AUTO_UPDATE_STATISTICS = ON default och procent samplad statistik kommer då automatiskt ändras efter ett antal modifikationer (exakt antal modifikationer beror på antal rader i tabellen).[1] ProjectionNested kan därför skilja i ungefär 1 sekunds exekveringstid beroende på denna variabel.

Ifrån klientens perspektiv är ändå ProjectionNested alltid snabbare eftersom IncludeEntities returnerar fler kolumner och materialiserar fullständiga entiteter. Därför blir varje rad bredare och en betydligt större mängd data överförs som Entity Framework Core behöver skapa objektgrafer åt. I ett fall då procent samplad statistik är hög hanterar SQL Server IncludeEntities mer effektivt än ProjectionNested, men kostnaden för att materialisera IncludeEntities i klienten leder i slutändan alltid till långsammare exekveringstid.

Hur kan hämtning av entiteter med komplexa relationer förbättras?

Tabell 6. BenchmarkDotNet resultat för AsSplitQuery jämfört med AsSingleQuery.

Metod	**Medelvärde exekveringstid**	**Allokerat minne**
ProjectionNested	1.9 s	388 MB
ProjectionNestedSplitQuery	1.2 s	308.1 MB
IncludeEntities	6.2 s	1249 MB
IncludeEntitiesSplitQuery	4.3 s	1061.9 MB

Se bilaga 2, sida 15-16, för de genererade SQL frågorna.

En stor fördel med AsSplitQuery() är att man undviker cartesian explosion. Entity Framework Core uppnår detta genom att generera separata SQL frågor för varje tabell den hämtar ifrån och använda INNER JOIN för att endast returnera kolumner från just den tabellen. Detta leder till minskad minnesanvändning i klienten.

ShowplanXML för dessa metoder visar på effektiviteten av AsSplitQuery(). SQL frågan för att hämta OrderItems står för största delen av exekveringstiden, men viktigt att notera är att när DegreeOfParallelism överstiger 1 är då CpuTime alltså summan av exekveringstid för parallela processer och den faktiska tiden för SQL Server att returnera ett svar kan vara betydligt kortare.

Tabell 7. Exekveringsplan värden per query för ProjectionNestedSplitQuery.

Query nummer	**Entitet**	**SQL Server CpuTime**	**DegreeOf Parallelism**
1	Customers	7 ms	1
2	Orders	333 ms	12
3	OrderItems	2454 ms	12

Tabell 8. Exekveringsplan värden per query för IncludeEntitiesSplitQuery.

Query nummer	**Entitet**	**CpuTime**	**DegreeOf Parallelism**
1	Customers	12 ms	1
2	Orders	473 ms	12
3	OrderItems	2659 ms	12

Slutsats och diskussion

Hur bör existens av objekt i en samling verifieras?

Resultaten har tydligt visat att användning av Any() ger stora fördelar mot Count() så länge entiteterna inte behöver materialiseras. När en applikation ska läsa från en databas i sammanhang som är känsliga för dessa logiska predikat bör frågan då ställas om entiteten verkligen behöver materialiseras eller om det går att svara på frågan genom andra metoder.

Kodexempel med materialisering.

```
If (db.Customers.Find(1).Orders.Any())
      Return true;
```

Kodexempel utan materialisering.

```
If (db.Orders.Any(x => x.CustomerId == 1))
      Return true;
```

Viktigt att notera är också att Entity Framework Core inte ser de yttre predikaten och därför inte översätter dem till SQL.

Kodexempel med ett yttre predikat.

```
Var hasCustomers = db.Customers.Count() > 0;
```

I ovan exempel kommer Entity Framework Core endast att fråga SQL Server efter antal rader i Customers tabellen medans jämförelsen mot 0 sker i ren C#.

Hur bör strängar jämföras?

Normalisering av strängar inom LINQ uttryck förhindrar SQL Server från att dra nytta av databasens Index. SQL Server stödjer även inställning för case-insensitive collation per kolumn. Om nödvändigt ska normalisering ske innan LINQ uttrycket formuleras.

Kodexempel med förnormaliserad sträng.

```
Var normalisedSku = productSku.ToLower()
Var product = db.Products.FirstOrDefault(x => x.Sku == normalisedSku);
```

Hur bör ett urval av fält från en eller flera tabeller materialiseras?

När det endast är nödvändigt att läsa ett urval av fält från en eller flera tabeller har det under undersökningen alltid lett till betydligt snabbare exekveringstid och mindre minnesallokering om materialisering sker genom att projicera till en smalare datatyp istället för att materialisera fullständiga entiteter.

Även i de fall när SQL Server har valt en mindre effektiv exekveringsplan för metoder som använder sig av projicering har det efter dataöverföring och materialisering ändå lett till bättre prestanda i klienten som konsumerar datan genom EF Core.

Projicerade datatyper plockas inte upp av EF Core's Tracker vilket måste tas hänsyn till om klienten även ska stödja modifiering av datan. För att dra nytta av fördelarna med projicering bör därför DTO:er användas när data ska modifieras.

Hur kan hämtning av entiteter med komplexa relationer förbättras?

Under undersökningen har användning av AsSplitQuery() alltid lett till snabbare exekveringstid och mindre minnesallokering i klienten.

En stark orsak för cartesian explosion är när den efterfrågade datan inkluderar flera relationer från samma nivå i objektgrafen och det är i dessa fall som användning av AsSplitQuery() kan ge störst prestandaförbättring.

AsSplitQuery() ska användas med åtanke för att det leder till att ökad trafik mellan servern och klienten eftersom flera separata SQL frågor måste hanteras. I vissa fall kan detta leda till försämrad prestanda.

När AsSplitQuery() används är det också möjligt att den berörda datan uppdateras mellan de separata SQL frågorna och klienten får då inkonsekvent data. För att motverka detta kan SQL frågorna wrappas i en Transaction men detta medför olika nackdelar beroende på hur det implementeras.[2]

Referenser

1. Microsoft, 18 November 2025, Statistics, Microsoft, https://learn.microsoft.com/en-us/sql/relational-databases/statistics/statistics?view=sql-server-ver17#auto_update_statistics-option, 19 Maj 2026
2. Microsoft, 25 Januari 2024, Single vs Split Queries, Microsoft, https://learn.microsoft.com/en-us/ef/core/querying/single-split-queries#characteristics-of-split-queries, 19 Maj 2026