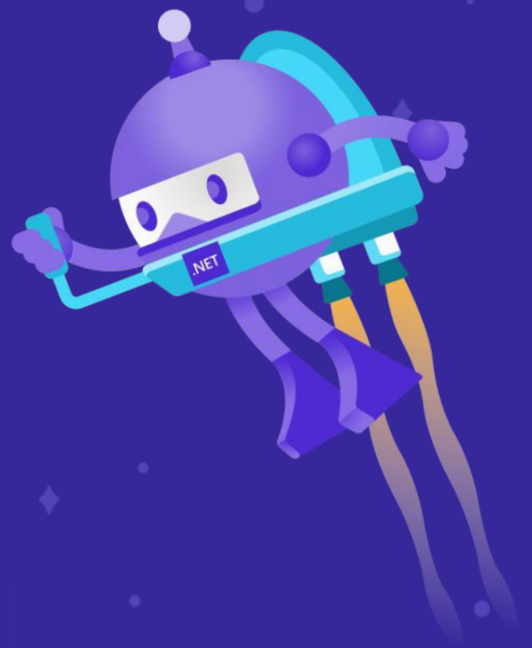
Entity Framework Core 효율적인 쿼리 사용

김정선 | ㈜씨퀄로 대표컨설턴트/이사
email: jskim@sqlroad.com



김정선 소개

SQL Server 컨설팅/연간기술지원/교육/솔루션 개발

SQL路

www.sqlroad.com 뉴스레터 신청 가능

www.sqlbigeyes.com

교육센터 / 기업체 출강 / 온라인 강의 & 멘토링

①	Entity Framework Core DB 성능 튜닝	인프런	
②	쿼리능력 레벨업(고급 T-SQL 쿼리)	인프런	
③	SW개발자를 위한 성능 좋은 쿼리 작성법	인프런	탈잉
④	SQL Server 대용량 데이터 처리 기술 이해	스킬서포트	
⑤	쿼리 튜닝 실무	스킬서포트	탈잉
⑥	DB 튜닝 실무	스킬서포트	
⑦	IT 발표/강연/강의 코칭 (멘토링)	인프런	

커뮤니티 & 대외 활동

- www.visualdb.net
- visualdb@Facebook
- Kim Jungsun@LinkedIn
- facebook.com/SQLPASSKorea
- Microsoft Data Platform MVP (since 2002)

세션의 취지

"EF Core에서 생성되는 모든 쿼리가 좋은 형태는 아니다."
by Microsoft

EF or EF Core 개발 시
DB/쿼리 성능을 고려한 구현

Java, JPA
Python, Django SQLAlchemy
Node.js, Sequelize

참고 사항

- 버전 기준
  - EF Core 7
  - SQL Server 2019

- 버전이나 SW에 따라 쿼리는 달라질 수 있습니다

가장 먼저 하고 싶은 말씀

"잘 작성된 LINQ 코드에서 좋은 SQL이 생성될 수 있다."

자동 생성되는 "쿼리 실체" 확인
성능 상 중요 쿼리는 "진단분석과 튜닝"

인덱스 튜닝

- 기본 키(PK)
  - 클러스터형 인덱스가 기본

    → 최선인지 검토 및 조정 (특히 대용량 테이블)
- 복합(다중 열) 인덱스 정의 시
  - 열 순서, 정렬 순서 고려
- 참조 키(FK) 대상 열
  - JOIN을 고려한 인덱스 생성
- Covering (또는 INCLUDE) 인덱스
  - OLAP 쿼리용으로 제한적 활용

동기 vs. 비동기 처리

- 비동기 처리 장점
  - 애플리케이션 확장성
    - DB 작업 시 스레드 차단 감소 (반응형 UI 처리 성능)
- 고려 사항
  - 개별 응답속도는 동기식보다 느릴 수 있음
  - 동기식 코드와 병행 사용은 비 추천(by MS, 내부 안정성)
- SqlClient 기준 알려진 Issue
  - https://github.com/dotnet/SqlClient/issues/593 (LargeOBject 열 검색 성능 저하, EF Core 동일)
  - https://github.com/dotnet/SqlClient/issues/601 (Connection 병렬 사용 시 성능 저하)
- 개발/테스트 단계에서 성능 확인 필요
  - 대량 결과 집합을 가지는 쿼리 동시 호출 시
  - 대량 트랜잭션 처리 시
  - 예상보다 훨씬 느린 응답 속도를 보이는 경우 (ex, 위 Issue)

```
await ComparetoASync();
……
static async Task ComparetoASync()
{
        ……
        var orders = await context.Orders
              .AsNoTracking()
              .Where(o => o.OrderKey > 1)
              .Take(500000)
              .ToListAsync()
              ;
         ……
```

참고. Benchmark - 동기 vs. 비동기 처리

- Issue Case - LOB열(ex, varchar(max)) 5MB 크기, 행 1건 단순 조회
  ① .ToList()
  ② .ToListAsync()

```
BenchmarkDotNet=v0.13.2, OS=Windows 11 (10.0.22621.963)
12th Gen Intel Core i7-12700H, 1 CPU, 20 logical and 14 physical cores
.NET SDK=7.0.101
  [Host]     : .NET 7.0.1 (7.0.122.56804), X64 RyuJIT AVX2  [AttachedDebugger]
  DefaultJob : .NET 7.0.1 (7.0.122.56804), X64 RyuJIT AVX2
```

주의. 실제와는 다를 수 있으므로 참고 용입니다.

Method	Mean	Error	StdDev	Median	Min	Max	Ratio	RatioSD
GetWithToList	15.93 ms	0.473 ms	1.351 ms	15.35 ms	14.40 ms	19.75 ms	1.00	0.00
GetWithToListAsync	247.12 ms	4.908 ms	13.352 ms	246.58 ms	226.00 ms	277.00 ms	15.62	1.65

비 추적(No Tracking) 쿼리 사용

Method	NumBlogs	NumPostsPerBlog	Mean	Error	StdDev	Median	Ratio	RatioSD	Gen 0	Gen 1	Gen 2
AsTracking	10	20	1,414 us	27.20 us	45.44 us	1,405.5 us	1.00	0.00	60.5469	13.6719	-
AsNoTracking	10	20	993 us	24.04 us	65.40 us	966.2 us	0.71	0.05	37.1094	6.8359	-

원본: MS Learn

- 이슈
  - 변경 사항 추적을 위해 쿼리 결과를 메모리 스냅상으로 저장
    - 메모리 소비, 처리 시간 소요
- 조치
  - 조회(읽기) 전용 쿼리는 "No Tracking"으로 사용
    - ① 쿼리 단위 설정
      - .AsNoTracking() vs. AsTracking()
      - .AsNoTrackingWithIdentityResolution() (EF Core 5)
    - ② Context 인스턴스 수준 설정
      - context.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking
    - ③ 전체 기본 동작으로 설정
      - optionBuilder.UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking)
      - DML(insert/update/delete) 작업 시 개별적으로 AsTracking() 적용

20만 건 조회 쿼리, 4회 측정 성능 (최대치로 기록)			
	Duration(ms)	CPU(ms)	Read(MB)
AsTracking	3,489	79	37
AsNoTracking	1,935	93	37

필요한 열 만 SELECT (Projection)

- 특히,
  - 데이터 대량 검색 (OLAP성 쿼리) 또는 대량 UPDATE 시
  - Large OBject 열을 제외한 테이블 검색
  - Covering Index (또는 INCLUDE 열) 적용 시 (ex. select name from dbo.customers where custkey = 1)

```
…… = await db.Customers
        .Where(c => c.Phone == phone)
        .AsNoTracking()
        .ToListAsync();
```

vs.

```
…… = await db.Customers
        .Where(c => c.Phone == phone)
        .Select(c => new {c.Name, c.CustKey})
        .AsNoTracking()
        .ToListAsync();
```

참고. Benchmark - 데이터 조회 방식 별 성능

- Blog 데이터 1,000건 조회 및 집계(sum, count)
  - ① 데이터 조회 후 EF Core에서 Loop로 행 단위 처리한 경우
  - ② .AsNoTracking() 적용한 경우
  - ③ Rating 열만 조회(Projection) 한 경우
  - ④ DB에서 모두 처리한 경우

```
BenchmarkDotNet=v0.13.2, OS=Windows 11 (10.0.22621.963)
12th Gen Intel Core i7-12700H, 1 CPU, 20 logical and 14 physical cores
.NET SDK=7.0.101
  [Host]     : .NET 7.0.1 (7.0.122.56804), X64 RyuJIT AVX2  [AttachedDebugger]
  DefaultJob : .NET 7.0.1 (7.0.122.56804), X64 RyuJIT AVX2
```

주의. 실제와는 다를 수 있으므로 참고 용입니다.

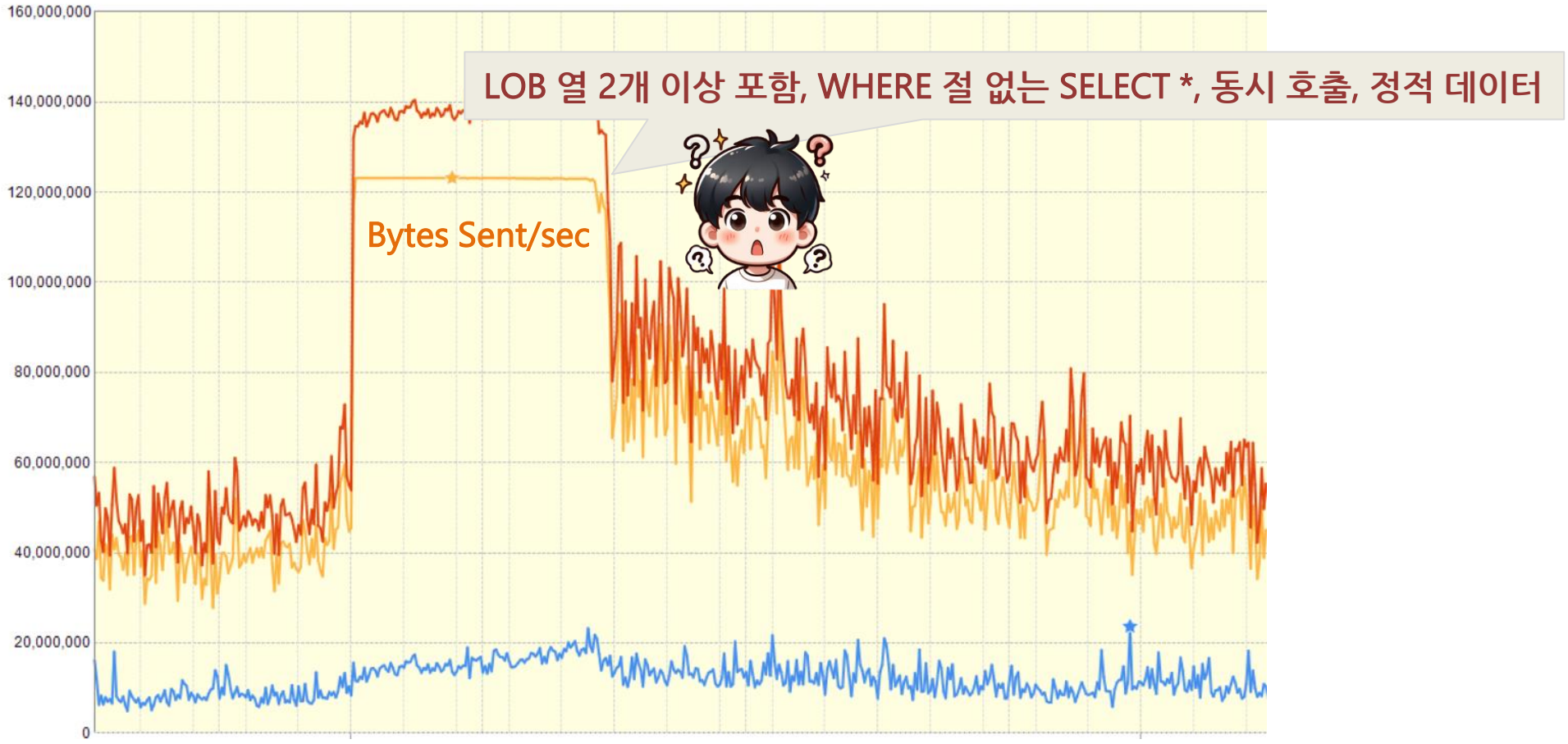
Method	NumBlogs	Mean	Error	StdDev	Ratio	RatioSD	Gen0	Gen1	Allocated	Alloc Ratio
LoadEntities	1000	2,090.2 µs	58.37 µs	168.41 µs	4.80	0.42	109.3750	42.9688	1385.96 KB	19.61
LoadEntitiesNoTracking	1000	1,048.9 µs	44.79 µs	132.07 µs	2.34	0.27	43.9453	2.9297	545.85 KB	7.72
ProjectOnlyRanking	1000	632.6 µs	16.00 µs	45.39 µs	1.45	0.12	20.5078	0.9766	258.13 KB	3.65
CalculateInDatabase	1000	438.8 µs	8.69 µs	22.43 µs	1.00	0.00	5.3711	-	70.67 KB	1.00

필요한 행만 검색 (Selection)

- 불필요한 대량 행 검색 부하
  - 대량 IO, 메모리 또는 CPU 부하
  - 네트워크 전송 부하
- [인덱스] 사용은 기본
- 구현
  ① Where() 에서 필터 조건으로 필요한 행 집합만 검색
  ② Single(), First() 등에서 필요한 필터 조건 처리
  ③ Find() 는 “TOP(1) + WHERE PK = @“로 처리
- 정적(불변) 데이터는 DB 반복 조회(특히 동시 대량) 부하 제거
  - Cache(ASP.NET Core, etc.) 기능 활용 권고

```
select * from table
```

사례 - DB서버 대량의 네트워크 대기



Buffering vs. Streaming

- LINQ 함수 별 IQueryable vs. IEnumerable 인터페이스
  - IQueryable(Streaming)
    - 최종 결과 호출 시 쿼리 실행
    - 쿼리 동적 구성 필요한 경우 유용
  - IEnumerable(Buffering)
    - 쿼리 실행 후 버퍼(메모리)에서 후속(ex. Where) 연산
      - 불필요한 (대량) 데이터 검색 부하 주의
    - 처리가 확정된 결과 검색 시 유용

```
static IQueryable<Order> GetOrdersQueryable()
{
      return (new SalesContext()).Orders;
}
static IEnumerable<Order> GetOrdersEnumable()
{
      return (new SalesContext()).Orders;
}
```

EventClass	ApplicationName	Duration	CPU	Reads	Writes	RowCounts	
RPC:Completed	Sales	1	0	19	0	1	A
SQL:BatchCompleted	Sales	14588	985	30135	0	1499999	B

```
var orders = GetOrdersQueryable()
if (p_orderkey.HasValue)
    orders = orders.Where(o => o.OrderKey == p_orderkey);
```

A

```
SELECT COUNT(*)
FROM [Orders] AS [o]
WHERE [o].[OrderKey] = @__p_orderkey_0
```

VS.

```
var orders = GetOrdersEnumable()
             .Where(o => o.OrderKey == 100);
```

B

```
SELECT [o].[OrderKey], ……
FROM [Orders] AS [o]
```

Lazy Loading vs. Eager Loading - 1/2

1+N?

*런타임에 동적으로 JOIN할 데이터를 결정할 경우 (일명 N+1 문제)

Lazy(지연)

```
List<Customer> customers = await db.Customers
      .Where(c => c.CustKey <= 5))
      .ToListAsync();

      foreach (Customer c in customers)
      {
         foreach (Order o in c.Orders)
         {
           Console.WriteLine($"OrderKey: {o.OrderKey}");
         }
      }
```

```
Executed DbCommand (6ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT [c].[CustKey], [c].[AcctBal], [c].[Address], [c].[Comment], [c].[MktSegment]
FROM [Customers] AS [c]
WHERE [c].[CustKey] <= 5
```

```
Executed DbCommand (1ms) [Parameters=[@__p_0='?' (DbType = Int32)]
Executed DbCommand (1ms) [Parameters=[@__p_0='?' (DbType = Int32)]
Executed DbCommand (1ms) [Parameters=[@__p_0='?' (DbType = Int32)]
Executed DbCommand (1ms) [Parameters=[@__p_0='?' (DbType = Int32)]
Executed DbCommand (1ms) [Parameters=[@__p_0='?' (DbType = Int32)]
SELECT [o].[OrderKey], [o].[Clerk], [o].[Comment], [o].[CustKey],
FROM [Orders] AS [o]
WHERE [o].[CustKey] = @__p_0
```

VS.

Join 처리 – *성능 상 이 방식을 기본으로 사용 권고*

Eager(동시)

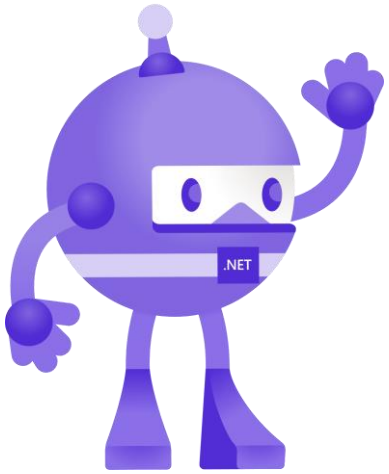
```
List<Customer> customers = await db.Customers
      .Include(c => c.Orders)
      .Where(c => c.CustKey <= 5))
      .ToListAsync();

      foreach (Customer c in customers)
      {
         foreach (Order o in c.Orders)
         {
           Console.WriteLine($"OrderKey: {o.OrderKey}");
         }
      }
```

```
Executed DbCommand (24ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT [c].[CustKey], [c].[AcctBal], [c].[Address], [c].[Comment], [c].[MktSegment],
FROM [Customers] AS [c]
LEFT JOIN [Orders] AS [o] ON [c].[CustKey] = [o].[CustKey]
WHERE [c].[CustKey] <= 5
ORDER BY [c].[CustKey]
```

Demonstration

- Lazy Loading vs. Eager Loading



사례 - 동일 쿼리 대량 반복 호출

TextData	SPID	EventSequence	StartTime	ApplicationName	EndTime
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.670	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.670	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.673	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.673	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.677	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.680	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.680	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.680	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.683	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.687	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.687	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.690	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.690	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.693	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.697	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.697	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.700	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.700	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.703	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.703	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.707	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.710	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.710	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.713	EntityFramework	:!
exec sp_executesql N'SELECT FROM (SELECT TOP (1)	93		08:42:54.713	EntityFramework	:!

Lazy Loading vs. Eager Loading - 2/2

A. Lazy Loading 사용 (필요한 경우 제한적으로 사용)
   - Microsoft.EntityFrameworkCore.Proxies 패키지
   - .UseLazyLoadingProxies(true)

B. Eager Loading 사용 (기본으로 사용)
   ① .Join()
      - INNER JOIN으로 처리
   ② .Include()
      - OUTER JOIN으로 처리
   ③ Projection (.Select()에서 Entity 참조, 일명 "Select Loading")
      - OUTER JOIN으로 처리
   ④ context.ChangeTracker.LazyLoadingEnabled = false
      - 참조 테이블이 검색(Query)되지 않음
   ⑤ 기타 ("Explicit Loading" 방법 등)

```
optionsBuilder
       .UseLazyLoadingProxies(true)
       .UseSqlServer(……)
```

```
services.AddDbContextPool……
       .UseLazyLoadingProxies(true)
       .UseSqlServer(……)
```

Include() 와 Split Queries - 1/2 - 문제

- 일명 “Cartesian Explosion” 문제
  - Join은 기본적으로 행 복제 연산 (중복 복제)
  - 1:N 관계 조인에서 NxM 결과(Cartesian)가 급증하는 경우 (중복 복제)
    - [1:2] 1x2 = 2, 5x2 = 10 vs. 5+2 = 7
    - [1:100] 1x100 = 100, 5x100 = 500(.Include) vs. 5 + 100 = 105(.AsSplitQuery)

N.id	N.name	N.?	M.id	M.qty	M.?
1	ABC	…	1	100	…
1	ABC	…	1	200	…
1	ABC	…	1	300	…
1	ABC	…	1	400	…
…	…	…	…	…	…

Include() 와 Split Queries - 2/2 - 조치 및 고려 사항

- Split Queries(쿼리 분할) 기능
  - 하나의 OUTER JOIN 쿼리를 개별 쿼리로 분리(1+1) 호출
    - EF Core 5에서 .Include용 도입
    - EF Core 6에서 .Include없이 Project용으로도 지원
  - 적용 대상
    - 조인 열 밀도(조인 행 수)가 매우 높은 경우 (높은 중복 수)
    - 1+1 분리 호출 시 성능 향상이 명백한 경우
  - 고려 사항
    - 실제론 분할 쿼리 수 만큼 중복 IO 발생
      - 인덱스 튜닝 여부에도 영향
    - 현재는 별도 배치로 DB 2번 호출 (차후 개선 예정)
      - 분리된 쿼리로 인해 결과 집합 일관성에도 영향 가능
    - ORDER BY 강제 발생은 동일하게 주의

전역 설정

```
sqloptions.UseQuerySplittingBehavior(QuerySplittingBehavior.SingleQuery)
```

개별 설정

```
orders = await db.Orders
            .Include(o => o.LineItems)
            .Where(o => o.CustKey == 1)
            .AsSplitQuery()
            .ToListAsync()
```

Internal Buffering

```
SELECT [o].[OrderKey], ......
FROM [Orders] AS [o]
WHERE [o].[CustKey] = 1
ORDER BY [o].[OrderKey]
go
SELECT [l].[OrderKey], ......, [o].[OrderKey]
FROM [Orders] AS [o]
INNER JOIN [LineItems] AS [l]
    ON [o].[OrderKey] = [l].[OrderKey]
WHERE [o].[CustKey] = 1
ORDER BY [o].[OrderKey]
```

EF Internal Buffering

- EF 내부적으로 필요 시 결과 집합을 내부 버퍼에 저장
  - ① Split Queries 사용 시
    - 마지막 쿼리 제외, MARS 비 사용 시
  - ② 후속 쿼리에서 이전 쿼리와 동일한 결과를 반환하는 경우
    - ex. 재시도(Retry) 전략

  - 주의
    - ToList() 등을 사용 시 연산자 자체 버퍼링 발생으로 중복 버퍼링 가능
      - https://learn.microsoft.com/en-us/ef/core/performance/efficient-querying

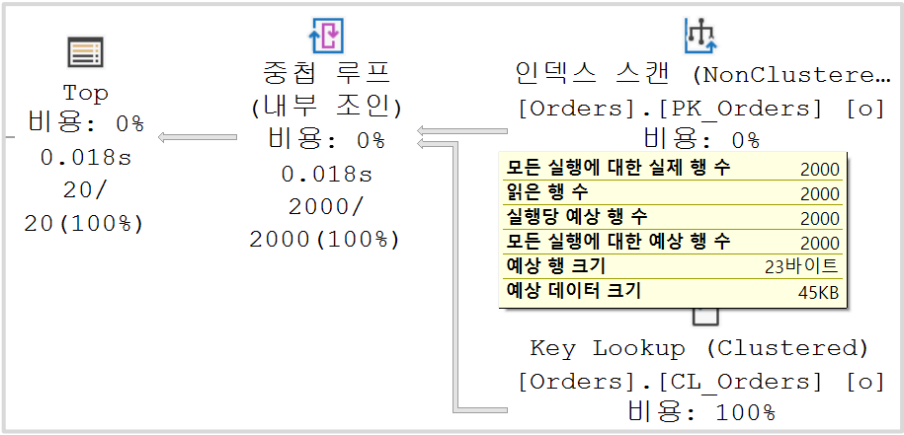
Paging 쿼리 - Skip() + Take()

- DB 성능의 큰 영향을 미치는 대표적 쿼리
- Paging 쿼리 어떻게 구현되나?
  - SQL Server 2012 이후 OFFSET FETCH로 구현
    - 이전엔 ROW_NUMBER() 로 구현
  - 매개변수화 지원
    - 과거 버전(EF 4?)은 리터럴로 처리
    - 현재(EF Core 7)는 매개변수로 처리
  - DB 측면의 튜닝 병행 필수
    - Index 튜닝
    - 쿼리 자체 튜닝
      - 필요한 데이터(행/열)만 DB에 요청

```
var orders = await db.Orders
        .AsNoTracking()
        .OrderByDescending(o => o.OrderKey)
        .Skip((currentpage - 1) * rowsperpage)
        .Take(rowsperpage)
        .ToListAsync()
```

currentpage:= 100
rowsperpage:= 20

```
exec sp_executesql N'SELECT [o].[OrderKey], ......
FROM [Orders] AS [o]
ORDER BY [o].[OrderKey] DESC
OFFSET @__p_0 ROWS FETCH NEXT @__p_1 ROWS ONLY',
N'@__p_0 int,@__p_1 int',@__p_0=1980,@__p_1=20
```



SQL 직접 사용 권고

① 성능 튜닝 및 관리가 필요한 OLAP성 쿼리

- 대용량 데이터 조회 및 연산 (ex, 대량조인/집계, 만능조회, 손익분석 등)
- 쿼리 힌트 등 복잡한 처리나 기능으로 쿼리 튜닝이 필요한 경우
- DB 단에서의 수정, 권한, 버전 등 관리 필요한 경우
- ORM 생성 쿼리에 성능 이슈가 있는 경우

② 대량 데이터 변경

- 대량 데이터 DML 작업
- DB 단에서의 Batch성 작업

SQL 직접 사용 - 함수 분류

- dbSet<>.FromSql *<entity>*(FormattableString sql) (EF Core 7)
  - dbSet<>.FromSqlRaw *<entity>*(string sql, params object[] parameters)
  - dbSet<>.FromSqlInterpolated *<entity>*(FormattableString sql)
- Task<int> ExecuteUpdateAsync() (EF Core 7)
  - int dbSet<>.ExecuteUpdate *<entity>*()
- Task<int> ExecuteDeleteAsync() (EF Core 7)
  - int dbSet<>.ExecuteDelete *<entity>*()
- Database.SqlQuery*<TResult>*(FormattableString sql) (EF Core 7)
  - Database.SqlQueryRaw*<TResult>*(string sql, params object[] parameters)
- Task<int> Database.ExecuteSqlAsync() (EF Core 7)
  - int Database.ExecuteSql(FormattableString sql)
  - Task<int> Database.ExecuteSqlRawAsync()
  - int Database.ExecuteSqlRaw(string sql, params object[] parameters) or Ienumerable object[] parameters
  - Task<int> Database.ExecuteSqlInterpolatedAsync()
  - int Database.ExecuteSqlInterpolated(FormattableString sql)

- 매개변수 쿼리
- 테이블 값 함수(TVFs)
- 스칼라 타입(Non-Entity) 쿼리
- Entity 기준 저장 프로시저

조회 전용(Non-Entity) 저장 프로시저 호출

- OLAP성 복합 쿼리용 프로시저
  - 대량 테이블 결합(Join, Subquery) 집합
  - 대량 쿼리 텍스트
  - SQL Server 특수 문법이나 기능 사용
  - DB 측면의 성능 튜닝이나 관리 필요
- 방법
  - Native SQL Client 프로그래밍
  - 혹은 Dapper.NET 등

```
var paramInfo = new[]
{
   new SqlParameter("@p_CustKey", SqlDbType.Int, 4) ……
   ……
};
using SalesContext db = new SalesContext();
using (var cnn = db.Database.GetDbConnection())
{
    var cmd = cnn.CreateCommand();
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.CommandText = "dbo.up_OrdersInfoByCustomer";
    cmd.Parameters.AddRange(paramInfo);
    cmd.Connection = cnn;

    await cnn.OpenAsync();
    var reader = await cmd.ExecuteReaderAsync(CommandBehavior.CloseConnection);
    if (reader.HasRows)
    {
        while (reader.Read())
        {
              ……
        }
        await reader.CloseAsync()
        Console.WriteLine($"{paramInfo[1].Value}, {paramInfo[2].Value}");
        ……
    }
……
```

참조 - 인프런 강의


.NET Entity Framework Core
for SQL Server
6개 무료 보기
NEW
개발 · 프로그래밍 > 데이터베이스
Entity Framework Core DB 성능 튜닝 (for SQL Server)
김정선SQL
.NET
EF Core

Q/A

Thank you!

김정선 @ 씨퀄로

Presented by aws elastic INFRAGISTICS Couchbase MVP .NET foundation