4.1.3 Подешавање перформанси

4.1.3.1 Lazy loading vs Eager loading

Entity Framework подразумевано користи Lazy Loading механизам за добављање података, што значи да се повезани подаци са циљаним ентитетом не преузимају одмах приликом добављања главног ентитета, већ по експлицитном захтеву. То се постиже тако што Entity Framework креира посредничке (енг. proxy) објекте, који, на захтев, комуницирају са базом како би добавили повезане податке који још нису преузети. Као што је већ било напоменуто, овај механизам може да изазове N+1 проблем, стога Entity Framework нуди могућност Eager Loading механизма. Ово се постиже коришћењем *Include* методе у LINQ израз, што се може видети у наредном примеру:

```
return Context.Users.Include(x => x.Posts).ToList();
```

Овим се у једном упиту учитавају и подаци о главном ентитету (User), као и о повезаном ентитету (Post). Међутим, треба још једном напоменути да прекомерно коришћење ове методе може негативно утицати на перформансе.

4.1.3.2 Индексирање и Профилисање

Ефикасно индексирање може драстично побољшати перформансе упита. Стварање индекса на колонама које се често користе у упитима или приликом операција спајања, може помоћи у бржем добављању података. Entity Framework нуди могућност креирања основних и композитних индекса над одговарајућим атрибутима ентитета и то се може извршити на два начина, уз помоћ анотација или коришћењем *FluentAPI-a*, посебног начина за конфигурацију модела, својственог за Entity Framework.

```
public class Tag
{
    public int TagId { get; set; }

    [Index]
    public string TagName { get; set; }

    public virtual ICollection<Post> Posts { get; set; } = new List<Post>();
}
```

- Слика 4.5 - Пример креирања индекса путем анотације

```
public partial class BenchmarkDbContext : DbContext
{
    public BenchmarkDbContext()
    {
    }

    public DbSet<User> Users { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<User>()
                    .Property(u => u.FirstName)
                    .HasColumnAnnotation("Index", new IndexAnnotation(new IndexAttribute("CompositeIndex", 1)));

        modelBuilder.Entity<User>()
                    .Property(u => u.LastName)
                    .HasColumnAnnotation("Index", new IndexAnnotation(new IndexAttribute("CompositeIndex", 2)));
    }
}
```

- Слика 4.6 - Пример креирања композитног индекса уз помоћ FluentAPI-а

Профилисање у Entity Framework алату представља битан инструмент за оптимизацију и надзор комуникације апликације са базом података. Омогућава развојним тимовима да идентификују неефикасне упите, пружа увид у преведене SQL упите и помаже у дијагностиковању потенцијалних проблема у перформансама. У својој основи, Entity Framework омогућава логовање путем којег се могу приказати информације о извршеном упиту, као што су време извршавања, добављена количина података, као и преведена LINQ наредба у нативни SQL. У примеру испод може се видети наредба постављена помоћу LINQ језика, као и стање лог фајла након извршења наредбе.

```
public List<User> GetUsersBornAfter(DateTime year)
{
  return Context.Users.Where(x => x.DateOfBirth >= year).ToList();
}
```

```
dbug: 7/26/2023 13:31:35.620 CoreEventId.QueryExecutionPlanned[10107] (Microsoft.EntityFrameworkCore.Query)
      Generated query execution expression:
      'queryContext => new SingleQueryingEnumerable<User>(
          (RelationalQueryContext)queryContext,
          RelationalCommandCache.QueryExpression(
              Projection Mapping:
              EmptyProjectionMember -> Dictionary<IProperty, int> { [Property: User.UserId (int) Required PK AfterSave:Throw, 0],
                                                                    [Property: User.DateOfBirth (DateTime?), 1],
                                                                    [Property: User.Email (string) MaxLength(100), 2],
                                                                    [Property: User.FirstName (string) MaxLength(50), 3],
                                                                    [Property: User.LastName (string) MaxLength(50), 4] }
              SELECT u.user_id, u.date_of_birth, u.email, u.first_name, u.last_name
              FROM Users AS u
              WHERE u.date_of_birth >= @__year_0),
              null,
              Func<QueryContext, DbDataReader, ResultContext, SingleQueryResultCoordinator, User>,
              EntityFramework_Benchmark.Settings.BenchmarkDbContext,
              False,
              False,
              True
      )'
```

4.1.3.3 Кеширање

Entity Framework омогућава неколико облика кеширања података:

1. Кеширање објеката (енг. Object caching) - Посебан *ObjectStateManager* објекат који је уграђен у контекст класу води рачуна објектима који су претходно добављени и налазе се у меморији. Ово је познатије и као кеширање првог нивоа (енг. First level cache)
2. Кеширање плана извршавања упита - Чување изгенерисане SQL наредбе за упите који се понављају неколико пута
3. Кеширање метаподатака - Служи за дељење метаподатака о моделу између неколико различитих конекција

Приликом кеширања првог нивоа, *ObjectStateManager* објекат води рачуна да ли се ентитет, који се тражи упитом, већ налази у меморији. Ако се у меморији налази ентитет са идентичним кључем који је требао да се добави, Entity Framework ће га укључити одмах у резултат упита. Треба напоменути да ће Entity Framework свакако извршити упит ка бази података, али ће се заобићити фаза материјализације објекта чиме се штеди на ресурсима. Са друге стране, треба узети у обзир да уколико се превише ентитета налази у меморији, то може негативно да утиче на перформансе упита.