**Sample DB with big data: https://github.com/Microsoft/sql-server-samples/releases**

**Table have big data:**

| **Table** | **Total Row** |  |
| --- | --- | --- |
| **[Person].[Person]** | **19972** |  |
| **[Sales].[Customer]** | **19820** |  |
| **[Person].[BusinessEntity]** | **20777** |  |
| **[Production].[BillOfMaterials]** | **2679** |  |
| **[Production].[WorkOrder]** | **72591** |  |
| **[Person].[BusinessEntityContact]** | **909** |  |
| **[Production].[Product]** | **504** |  |

**https://github.com/Microsoft/sql-server-samples/releases**

**Slide 1:**

**Title:** Optimizing Entity Framework Core for Performance

**Subtitle:** Boosting Your Data Access Efficiency

**Image:** A speedometer gauge with the needle pointing to "High Performance"

**Slide 2:**

**Key Areas of Optimization:**

* Query Performance
* Change Tracking
* Loading Strategies
* Caching
* Connection Management

**Image:** A checklist with checkmarks next to each optimization area

**Slide 3:**

**Query Performance:**

* **Example:** Using Include and Select judiciously
* **Code demo:** Two code blocks, one with inefficient query and one with optimized query
* **Benchmark:** Graph showing query execution time reduction

**Image:** A magnifying glass focusing on a SQL query

**Slide 4:**

**Change Tracking:**

* **Example:** Disabling tracking when not needed
* **Code demo:** Code snippet demonstrating change tracking disablement
* **Benchmark:** Graph showing performance gains for bulk operations

**Image:** A toggle switch labeled "Change Tracking"

**Slide 5:**

**Loading Strategies:**

* **Example:** Choosing Eager Loading, Lazy Loading, or Explicit Loading
* **Code demo:** Code examples for each loading strategy
* **Image:** A flowchart illustrating loading strategy decision-making

**Slide 6:**

**Caching:**

* **Example:** Implementing query caching with EF Core Second Level Caching
* **Code demo:** Code setting up second-level caching
* **Benchmark:** Graph showing performance improvements for frequent queries

**Image:** A cache storage box with lightning bolts symbolizing speed

**Slide 7:**

**Connection Management:**

* **Example:** Using a connection pool for efficient database connections
* **Code demo:** Code snippet demonstrating connection pooling
* **Benchmark:** Chart showing reduced connection overhead

**Image:** A pool of database connections with multiple users accessing them

**Slide 8:**

**Additional Optimization Tips:**

* Indexing database tables effectively
* Analyzing query plans
* Updating EF Core to the latest version
* Using tools like EF Core Profiler

**Image:** A toolbox with optimization tools

**Slide 9:**

**Conclusion:**

* Apply these techniques to enhance EF Core performance significantly.
* Measure and benchmark to ensure optimal results.
* Embrace best practices for efficient and scalable data access layers.

**Image:** A rocket ship taking off, symbolizing performance boost

**Remember to:**

* Use clear and concise language.
* Highlight key points with visual cues.
* Maintain a consistent design theme.
* Practice your presentation to ensure smooth delivery.

—--------------------------------------------------------------------------

Version 2

Slide 1: Introduction

* Title: Entity Framework Core Optimization: Unleashing Performance
* Subtitle: Practical Techniques for Faster Data Access
* Speaker: [Your Name]
* Affiliation: [Your Company or Organization]

Slide 2: What is EF Core?

* Object-relational mapper (ORM) for .NET
* Simplifies data access and interactions with databases
* Key features:
  + Code-first approach
  + LINQ support for queries
  + Change tracking
  + Migrations

Slide 3: Why Optimize EF Core?

* Improve application performance
* Reduce database load
* Enhance user experience
* Minimize resource usage

Slide 4: Key Optimization Areas

* Database Design and Relationships
* Query Optimization
* Tracking and Change Detection
* Caching and Precomputed Data
* Performance Monitoring and Profiling

Slide 5: Database Design and Relationships (Demo Code)

* Example: Proper indexing for efficient queries:

C#

modelBuilder.Entity<Product>()

.HasIndex(p => p.CategoryId)

.IsUnique();

Slide 6: Query Optimization (Demo Code)

Query Optimization

* LINQ Fundamentals for Efficient Queries:
  + Demonstrate basic LINQ syntax for filtering, projections, and joins.
  + Emphasize the importance of understanding query translation to SQL.
* Eager Loading vs. Lazy Loading:
  + When to use each strategy for optimal data retrieval.
  + Code example:
  + C#
* Explicit Loading for Fine-Grained Control:
  + Use Load() for explicit loading of related entities.
* Avoiding Redundant Data Fetching:
  + Select only necessary columns with projections.
  + Filter data in the query instead of after retrieval.
  + Use AsNoTracking() for read-only queries.
* Optimizing Joins:
  + Analyze query plans to identify inefficient joins.
  + Restructure queries or use alternative approaches if necessary.
* Stored Procedures (Limited Use in EF Core):
  + Discuss their potential benefits and drawbacks in EF Core.
* Example: Eager loading related entities:

C#

var products = await context.Products

.Include(p => p.Category)

.ToListAsync();

* Example: Using AsNoTracking() for read-only queries:

C#

var productDetails = await context.Products

.AsNoTracking()

.Where(p => p.Id == productId)

.SingleOrDefaultAsync();

Slide 7: Tracking and Change Detection (Demo Code)

* Understanding Change Tracking:
  + Explain how EF Core tracks entity changes.
  + Impact of tracking on performance.
* Explicit Change Tracking for Control:
  + Use ChangeTracker for advanced scenarios.
* Disabling Automatic Change Detection for Large Entities:
  + context.ChangeTracker.AutoDetectChangesEnabled = false;
* Example: Disabling change tracking for specific entities:

C#

context.Entry(product).State = EntityState.Unchanged;

Slide 8: Caching and Precomputed Data (Demo Code)

* Example: Implementing query caching:

C#

var cachedProducts = await context.Products

.Cacheable()

.ToListAsync();

https://github.com/SteffenMangold/EntityFrameworkCore.Cacheable

Slide 9: Performance Monitoring and Profiling

* EF Core logging and diagnostic tools
* Query plan analysis tools
* Application profiling techniques

Slide 10: Best Practices and Tips

* Lightweight Models:
  + Avoid excessive entity projections.
* Batch Operations:
  + Use for bulk inserts/updates.
* Normalization vs. Denormalization:
  + Choose based on access patterns and query complexity.
* Execution Strategies:
  + Select appropriate tracking/no-tracking mode for each query.
* Monitoring and Continuous Improvement:
  + Use EF Core logging and diagnostic tools.
  + Profile application performance to identify bottlenecks.

Slide 11:

* Explicit Loading for Fine-Grained Control:
  + Use Load() for explicit loading of related entities.

C#

Copy

using (var context = new BloggingContext())

{

var blog = context.Blogs

.Single(b => b.BlogId == 1);

context.Entry(blog)

.Collection(b => b.Posts)

.Load();

context.Entry(blog)

.Reference(b => b.Owner)

.Load();

}

* Avoiding Redundant Data Fetching:
  + Select only necessary columns with projections.
  + Filter data in the query instead of after retrieval.
  + Use AsNoTracking() for read-only queries.
* Optimizing Joins:
  + Analyze query plans to identify inefficient joins.
  + Restructure queries or use alternative approaches if necessary.
* Stored Procedures (Limited Use in EF Core):
  + Discuss their potential benefits and drawbacks in EF Core.

slide 12 Using compile query

Pros:

* Performance:
  + Significant performance gains, especially for frequently executed queries.
  + Reduced query translation overhead at runtime.
  + Optimized SQL generation and execution by the database.
* Security:
  + Can help mitigate SQL injection attacks by pre-compiling queries with parameterized values.
* Error Detection:
  + Compile-time error checking for invalid query syntax or type mismatches.
* Caching:
  + Compiled queries can be cached more effectively for reuse.

Cons:

* Setup Complexity:
  + Initial setup and configuration can be more involved.
  + Requires defining queries as strings or using query tags.
* Maintenance:
  + Changes to queries or models might require re-compilation.
  + Rebuilding the application often necessary to update compiled queries.
* Flexibility:
  + Less flexible for dynamic queries or those built at runtime.
  + May not be suitable for ad-hoc queries or those with highly variable parameters.
* Debugging:
  + Debugging compiled queries can be more challenging, as errors might not be caught at compile time.

When to Use Compiled Queries:

* Frequently executed queries: High performance gains justify setup and maintenance overhead.
* Security-critical queries: Pre-compilation helps prevent SQL injection vulnerabilities.
* Queries with complex logic or joins: Can benefit from optimized SQL generation and execution.

Considerations:

* Balance performance and flexibility: Evaluate the trade-offs based on your application's needs.
* Use judiciously: Compile only frequently used or sensitive queries.
* Combine with other optimization techniques: Compiled queries can complement other EF Core performance strategies.
* Monitor and adjust: Continuously evaluate the effectiveness of compiled queries and make adjustments as needed.

Demo

using System.Linq;

using Microsoft.EntityFrameworkCore;

// ... (other code)

// 1. Define the compiled query as a string

string compiledQuery = @"

SELECT \*

FROM Products

WHERE CategoryId = @categoryId

";

// 2. Compile the query using the DbContext

var compiledQuery2 = context.Database.CompileQuery(compiledQuery);

// 3. Execute the compiled query with parameters

var products = compiledQuery2

.AsNoTracking() // Optional for read-only queries

.Parameters(new { categoryId = 1 })

.ToList();

// ... (use the retrieved products)

