From: "Milan Jovanović"

To:bensalahamine546@gmail.com

Cc:

Subject:📧 Fast SQL Bulk Inserts With C# and EF Core

Body:

** Fast SQL Bulk Inserts With C# and EF Core ** Read on: my website (https://click.convertkit-mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk/25h2hoh2nvgqe8c3/aHR0cHM6Ly93d3cubWlsYW5qb3Zhbm92aWN
) / Read time: 8 minutes BROUGHT TO YOU BY (https://click.convertkit-mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk/qvh8h7hrq7vl0vsl/aHR0cHM6Ly93d3cucG9zdG1hbi5jb20vcG9zdG
)The Best API Conference ----------------------- Get ready for POST/CON 24 (https://click.convertkit-mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk/qvh8h7hrq7vl0vsl/aHR0cHM6Ly93d3cucG9zdG1hbi5jb20vcG9zdG
)! Join us in San Francisco from April 30 - May 1 for Postman's biggest API conference ever. Use PC24DR100ALLIN to get 50% original price for all access. -->Register Here! (https://click.convertkit-mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk/g3hnh5h3xdn0koar/aHR0cHM6Ly9wb3N0Y29uLnBvc3RtYW4uY29
) Register Here! (https://postcon.postman.com/event/5bb77b40-684b-4b79-bb8f-f1420897ee6e/register?utm_source=influencer&utm_medium=Social&utm_campaign=POSTCON_2024&utm_term=Milan_Jovanovic&
) TOGETHER WITH (https://click.convertkit-mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk/9qhzhnhpnqxellu9/aHR0cHM6Ly9zaGVzaGEuaW8vP3V0bV9zb3V
)Low-Code Framework for .NET Devs ------------------------------- Introducing Shesha (https://click.convertkit-mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk/9qhzhnhpnqxellu9/aHR0cHM6Ly9zaGVzaGEuaW8vP3V0bV9zb3V
), a brand new, open-source, low-code framework for .NET developers. With Shesha, you can create business applications faster and with >80% less code! -->Learn More Here! (https://click.convertkit-mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk/9qhzhnhpnqxellu9/aHR0cHM6Ly9zaGVzaGEuaW8vP3V0bV9zb3V
) Learn More Here! (https://shesha.io/?utm_source=milan&utm_medium=email&utm_campaign=20240323) Whether you're building a data analytics platform, migrating a legacy system, or onboarding a surge of new users, there will likely come a time when you'll need to insert a massive amount of data into your database. Inserting the records one by one feels like watching paint dry in slow motion. Traditional methods won't cut it. So, understanding fast bulk insert techniques with C# and EF Core becomes essential. In today's issue, we'll explore several options for performing bulk inserts in C#: * Dapper * EF Core * EF Core Bulk Extensions * SQL Bulk Copy The examples are based on a User class with a respective Users table in SQL Server. public class User { public int Id { get; set; } public string Email { get; set; } public string FirstName { get; set; } public string LastName { get; set; } public string PhoneNumber { get; set; } } This isn't a complete list of bulk insert implementations. There are a few options I didn't explore, like manually generating SQL statements and using Table-Valued parameters. --------------------- EF Core Naive Approach --------------------- Let's start with a simple example using EF Core. We're creating an ApplicationDbContext instance, adding a User object, and calling SaveChangesAsync. This will insert each record to the database one by one. In other words, each record requires one round trip to the database. using var context = new ApplicationDbContext(); foreach (var user in GetUsers())**

{ context.Users.Add(user); await context.SaveChangesAsync(); } The results are as poor as you'd expect: EF Core - Add one and save, for 100 users: 20 ms EF Core - Add one and save, for 1,000 users: 260 ms EF Core - Add one and save, for 10,000 users: 8,860 ms I omitted the results with 100,000 and 1,000,000 records because they took too long to execute. We'll use this as a "how not to do bulk inserts" example. ---------- Dapper Simple Insert ----------Dapper (https://click.convertkit-mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk/3ohphkh7vorkndir/aHR0cHM6Ly9naXRodWIuY29tL0RhcHBlckxp) is a simple SQL-to-object mapper for .NET. It allows us to easily insert a collection of objects into the database. I'm using Dapper's feature to unwrap a collection into a SQL INSERT statement. using var connection = new SqlConnection(connectionString); connection.Open(); const string sql = @" INSERT INTO Users (Email, FirstName, LastName, PhoneNumber) VALUES (@Email, @FirstName, @LastName, @PhoneNumber); "; await connection.ExecuteAsync(sql, GetUsers()); The results are much better than the initial example: Dapper - Insert range, for 100 users: 10 ms Dapper - Insert range, for 1,000 users: 113 ms Dapper - Insert range, for 10,000 users: 1,028 ms Dapper - Insert range, for 100,000 users: 10,916 ms Dapper - Insert range, for 1,000,000 users: 109,065 ms ---------- EF Core Add and Save ---------- However, EF Core still didn't throw in the towel. The first example was poorly implemented on purpose. EF Core can batch multiple SQL statements together, so let's use that. If we make a simple change, we can get significantly better performance. First, we're adding all the objects to the ApplicationDbContext. Then, we're going to call SaveChangesAsync only once. EF will create a batched SQL statement - group many INSERT statements together - and send them to the database together. This reduces the number of round trips to the database, giving us improved performance. using var context = new ApplicationDbContext(); foreach (var user in GetUsers()) { context.Users.Add(user); } await context.SaveChangesAsync(); Here are the benchmark results of this implementation: EF Core - Add all and save, for 100 users: 2 ms EF Core - Add all and save, for 1,000 users: 18 ms EF Core - Add all and save, for 10,000 users: 203 ms EF Core - Add all and save, for 100,000 users: 2,129 ms EF Core - Add all and save, for 1,000,000 users: 21,557 ms Remember, it took Dapper 109 seconds to insert 1,000,000 records. We can achieve the same with EF Core batched queries in ~21 seconds. ---------- EF Core AddRange and Save ---------- This is an alternative to the previous example. Instead of calling Add for all objects, we can call AddRange and pass in a collection. I wanted to show this implementation because I prefer it over the previous one. using var context = new ApplicationDbContext(); context.Users.AddRange(GetUsers()); await context.SaveChangesAsync(); The results are very similar to the previous example: EF Core - Add range and save, for 100 users: 2 ms EF Core - Add range and save, for 1,000 users: 18 ms EF Core - Add range and save, for 10,000 users: 204 ms EF Core - Add range and save, for 100,000 users: 2,111 ms EF Core - Add range and save, for 1,000,000 users: 21,605 ms ---------- EF Core Bulk Extensions ---------- There's an awesome library called EF Core Bulk Extensions (https://click.convertkit-mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk/n2hohvh3e8glknu6/aHR0cHM6Ly9naXRodWIuY29tL2JvcmlzZGov) that we can use to squeeze out more performance. You can do a lot more than bulk inserts with this library, so I recommend exploring it. This library is open source, and has a community license if you meet the free usage criteria. Check the licensing section (https://click.convertkit-mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk/48hvhehrk597nrix/aHR0cHM6Ly9naXRodWIuY29tL2JvcmlzZGovl) for more details. For our use case, the BulkInsertAsync method is an excellent choice. We can pass the collection of objects, and it will perform an SQL bulk insert. using var context = new ApplicationDbContext(); await context.BulkInsertAsync(GetUsers()); The performance is equally amazing: EF Core - Bulk Extensions, for 100 users: 1.9 ms EF Core - Bulk Extensions, for 1,000 users: 8 ms EF Core - Bulk Extensions, for 10,000 users: 76 ms EF Core - Bulk Extensions, for 100,000 users: 742 ms EF Core - Bulk Extensions, for 1,000,000 users: 8,333 ms For comparison, we needed ~21 seconds to insert 1,000,000 records with EF Core batched queries. We can do the same with the Bulk Extensions (https://click.convertkit-mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk/n2hohvh3e8glknu6/aHR0cHM6Ly9naXRodWIuY29tL2JvcmlzZGov) library in just 8 seconds. ---------- SQL Bulk Copy ---------- Lastly, if we can't get the desired performance from EF Core, we can try using SqlBulkCopy. SQL Server supports bulk copy operations (https://click.convertkit-

mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk/wnh2hghw854l3qh7/aHR0cHM6Ly9sZWFybi5taWNyb3NvZnQuY2
) natively, so let's use this. This implementation is slightly more complex than the EF Core examples. We need to configure the SqlBulkCopy instance and create a DataTable containing the objects we want to insert. using var bulk = new SqlBulkCopy(ConnectionString); bulk.DestinationTableName = "dbo.Users"; bulk.ColumnMappings.Add(nameof(User.Email), "Email"); bulk.ColumnMappings.Add(nameof(User.FirstName), "FirstName"); bulk.ColumnMappings.Add(nameof(User.LastName), "LastName"); bulk.ColumnMappings.Add(nameof(User.PhoneNumber), "PhoneNumber"); await bulk.WriteToServerAsync(GetUsersDataTable()); However, the performance is blazing fast: SQL Bulk Copy, for 100 users: 1.7 ms SQL Bulk Copy, for 1,000 users: 7 ms SQL Bulk Copy, for 10,000 users: 68 ms SQL Bulk Copy, for 100,000 users: 646 ms SQL Bulk Copy, for 1,000,000 users: 7,339 ms Here's how you can create a DataTable and populate it with a list of objects: DataTable GetUsersDataTable() { var dataTable = new DataTable(); dataTable.Columns.Add(nameof(User.Email), typeof(string)); dataTable.Columns.Add(nameof(User.FirstName), typeof(string)); dataTable.Columns.Add(nameof(User.LastName), typeof(string)); dataTable.Columns.Add(nameof(User.PhoneNumber), typeof(string)); foreach (var user in GetUsers()) { dataTable.Rows.Add(user.Email, user.FirstName, user.LastName, user.PhoneNumber); } return dataTable; } ----- Results ------- Here are the results for all the bulk insert implementations: | Method | Size | Speed |---------- |---------- |----------: | EF_OneByOne | 100 | 19.80 ms | | EF_OneByOne | 1000 | 259.87 ms | | EF_OneByOne | 10000 | 8,860.79 ms | | EF_OneByOne | 100000 | N/A | | EF_OneByOne | 1000000 | N/A | | Dapper_Insert | 100 | 10.650 ms | | Dapper_Insert | 1000 | 113.137 ms | | Dapper_Insert | 10000 | 1,027.979 ms | | Dapper_Insert | 100000 | 10,916.628 ms | | Dapper_Insert | 1000000 | 109,064.815 ms | | EF_AddAll | 100 | 2.064 ms | | EF_AddAll | 1000 | 17.906 ms | | EF_AddAll | 10000 | 202.975 ms | | EF_AddAll | 100000 | 2,129.370 ms | | EF_AddAll | 1000000 | 21,557.136 ms | | EF_AddRange | 100 | 2.035 ms | | EF_AddRange | 1000 | 17.857 ms | | EF_AddRange | 10000 | 204.029 ms | | EF_AddRange | 100000 | 2,111.106 ms | | EF_AddRange | 1000000 | 21,605.668 ms | | BulkExtensions | 100 | 1.922 ms | | BulkExtensions | 1000 | 7.943 ms | | BulkExtensions | 10000 | 76.406 ms | | BulkExtensions | 100000 | 742.325 ms | | BulkExtensions | 1000000 | 8,333.950 ms | | BulkCopy | 100 | 1.721 ms | | BulkCopy | 1000 | 7.380 ms | | BulkCopy | 10000 | 68.364 ms | | BulkCopy | 100000 | 646.219 ms | | BulkCopy | 1000000 | 7,339.298 ms | -------- Takeaway -------- SqlBulkCopy holds the crown for maximum raw speed. However, EF Core Bulk Extensions (https://click.convertkit-
mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk/n2hohvh3e8glknu6/aHR0cHM6Ly9naXRodWIuY29tL2JvcmlzZGov
) deliver fantastic performance while maintaining the ease of use that Entity Framework Core is known for. The best choice hinges on your project's specific demands: * Performance is all that matters? SqlBulkCopy is your solution. * Need excellent speed and streamlined development? EF Core is a smart choice. I leave it up to you to decide which option is best for your use case. Hope this was helpful. See you next week.Milan Jovanović ---------- Find me online:X/Twitter (https://click.convertkit-
mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk/reh8hoh0n27oewa2/aHR0cHM6Ly90d2l0dGVyLmNvbS9tam92YW5
) / LinkedIn (https://click.convertkit-
mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk/08hwh9hdqoxpkzal/aHR0cHM6Ly93d3cubGlua2VkaW4uY29tL2luI
) / YouTube (https://click.convertkit-
mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk/8ghqhohlk037xkfk/aHR0cHM6Ly93d3cueW91dHViZS5jb20vQE1pb
)P.S. There are 2 ways I can help you improve your skills: (https://click.convertkit-
mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk/vqh3hrhp8x4qdmhg/aHR0cHM6Ly93d3cubWlsYW5qb3Zhbm92aW
) ---------- Modular Monolith Architecture ---------- This in-depth course will transform the way you build monolith systems. You will learn the best practices for applying the Modular Monolith architecture in a real-world scenario. -->JOIN THE WAITLIST (https://click.convertkit-
mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk/vqh3hrhp8x4qdmhg/aHR0cHM6Ly93d3cubWlsYW5qb3Zhbm92aW
) JOIN THE WAITLIST (https://www.milanjovanovic.tech/modular-monolith-architecture?
utm_source=newsletter&utm_medium=email&utm_campaign=tnw82) (https://click.convertkit-
mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk/l2hehmh3w04dzeb6/aHR0cHM6Ly93d3cubWlsYW5qb3Zhbm92aW

) ---------- Pragmatic Clean Architecture ---------- This is the complete blueprint for building production-ready applications using Clean Architecture. Join 2,500+ other students to accelerate your growth as a software architect. -->START WATCHING NOW (https://click.convertkit-mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk/l2hehmh3w04dzeb6/aHR0cHM6Ly93d3cubWlsYW5qb3Zhbm92aW) START WATCHING NOW (https://www.milanjovanovic.tech/pragmatic-clean-architecture?utm_source=newsletter&utm_medium=email&utm_campaign=tnw82) BEFORE YOU LEAVE 👋 How am I doing? ---------- I love hearing from readers, and I'm always looking for feedback. How am I doing with The .NET Weekly? Is there anything you'd like to see more or less of? Which aspects of the newsletter do you enjoy the most? Hit reply and say hello - I'd love to hear from you! Stay awesome, Milan You received this email because you subscribed to our list. You can unsubscribe (https://unsubscribe.convertkit-mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk) at any time.Update your profile (https://preferences.convertkit-mail.com/68uwe4lvzpt8hkexrxqbohzdqrokk) | Dragiše Cvetkovića 2, Niš, - 18000

The .NET Weekly

Fast SQL Bulk Inserts With C# and EF Core

Read on: **my website** / Read time: **8 minutes**

business applications faster and with >80% less code!

Learn More Here!

Whether you're building a data analytics platform, migrating a legacy system, or onboarding a surge of new users, there will likely come a time when you'll need to insert a massive amount of data into your database.

Inserting the records one by one feels like watching paint dry in slow motion. Traditional methods won't cut it.

So, understanding fast bulk insert techniques with C# and EF Core becomes essential.

In today's issue, we'll explore several options for performing bulk inserts in C#:

- Dapper
- EF Core
- EF Core Bulk Extensions
- SQL Bulk Copy

The examples are based on a **User** class with a respective **Users** table in **SQL Server**.

```
public class User
{
    public int Id { get; set; }
    public string Email { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string PhoneNumber { get; set; }
}
```

This isn't a complete list of bulk insert implementations. There are a few options I didn't explore, like manually generating SQL statements and using Table-Valued parameters.

EF Core Naive Approach

Let's start with a simple example using EF Core. We're creating an `ApplicationDbContext` instance, adding a `User` object, and calling `SaveChangesAsync`. This will insert each record to the database one by one. In other words, each record requires one round trip to the database.

```
using var context = new ApplicationDbContext();

foreach (var user in GetUsers())
{
    context.Users.Add(user);

    await context.SaveChangesAsync();
}
```

The results are as poor as you'd expect:

```
EF Core - Add one and save, for 100 users: 20 ms
EF Core - Add one and save, for 1,000 users: 260 ms
EF Core - Add one and save, for 10,000 users: 8,860 ms
```

I omitted the results with **100,000** and **1,000,000** records because they took too long to execute.

We'll use this as a "how not to do bulk inserts" example.

Dapper Simple Insert

Dapper is a simple SQL-to-object mapper for .NET. It allows us to easily insert a collection of objects into the database.

I'm using Dapper's feature to unwrap a collection into a SQL **INSERT** statement.

```
using var connection = new SqlConnection(connectionString);
connection.Open();

const string sql =
    @"
    INSERT INTO Users (Email, FirstName, LastName, PhoneNumber)
    VALUES (@Email, @FirstName, @LastName, @PhoneNumber);
    ";

await connection.ExecuteAsync(sql, GetUsers());
```

The results are much better than the initial example:

```
Dapper - Insert range, for 100 users: 10 ms
Dapper - Insert range, for 1,000 users: 113 ms
Dapper - Insert range, for 10,000 users: 1,028 ms
Dapper - Insert range, for 100,000 users: 10,916 ms
Dapper - Insert range, for 1,000,000 users: 109,065 ms
```

EF Core Add and Save

However, EF Core still didn't throw in the towel. The first example was poorly implemented on purpose. EF Core can batch multiple SQL

statements together, so let's use that.

If we make a simple change, we can get significantly better performance. First, we're adding all the objects to the **ApplicationDbContext**. Then, we're going to call **SaveChangesAsync** only once.

EF will create a batched SQL statement - group many **INSERT** statements together - and send them to the database together. This reduces the number of round trips to the database, giving us improved performance.

```
using var context = new ApplicationDbContext();

foreach (var user in GetUsers())
{
    context.Users.Add(user);
}

await context.SaveChangesAsync();
```

Here are the benchmark results of this implementation:

```
EF Core - Add all and save, for 100 users: 2 ms
EF Core - Add all and save, for 1,000 users: 18 ms
EF Core - Add all and save, for 10,000 users: 203 ms
EF Core - Add all and save, for 100,000 users: 2,129 ms
EF Core - Add all and save, for 1,000,000 users: 21,557 ms
```

Remember, it took Dapper **109 seconds** to insert `1,000,000` records. We can achieve the same with EF Core batched queries in **~21 seconds**.

EF Core AddRange and Save

This is an alternative to the previous example. Instead of calling **Add** for all objects, we can call **AddRange** and pass in a collection.

I wanted to show this implementation because I prefer it over the previous one.

```
using var context = new ApplicationDbContext();

context.Users.AddRange(GetUsers());

await context.SaveChangesAsync();
```

The results are very similar to the previous example:

```
EF Core - Add range and save, for 100 users: 2 ms
EF Core - Add range and save, for 1,000 users: 18 ms
EF Core - Add range and save, for 10,000 users: 204 ms
```

```
EF Core - Add range and save, for 100,000 users: 2,111 ms
EF Core - Add range and save, for 1,000,000 users: 21,605 ms
```

EF Core Bulk Extensions

There's an awesome library called EF Core Bulk Extensions that we can use to squeeze out more performance. You can do a lot more than bulk inserts with this library, so I recommend exploring it. This library is open source, and has a community license if you meet the free usage criteria. Check the licensing section for more details.

For our use case, the **BulkInsertAsync** method is an excellent choice. We can pass the collection of objects, and it will perform an SQL bulk insert.

```
using var context = new ApplicationDbContext();

await context.BulkInsertAsync(GetUsers());
```

The performance is equally amazing:

```
EF Core - Bulk Extensions, for 100 users: 1.9 ms
EF Core - Bulk Extensions, for 1,000 users: 8 ms
EF Core - Bulk Extensions, for 10,000 users: 76 ms
EF Core - Bulk Extensions, for 100,000 users: 742 ms
EF Core - Bulk Extensions, for 1,000,000 users: 8,333 ms
```

For comparison, we needed **~21 seconds** to insert `1,000,000` records with EF Core batched queries. We can do the same with the Bulk Extensions library in just **8 seconds**.

SQL Bulk Copy

Lastly, if we can't get the desired performance from EF Core, we can try using **SqlBulkCopy**. SQL Server supports bulk copy operations natively, so let's use this.

This implementation is slightly more complex than the EF Core examples. We need to configure the **SqlBulkCopy** instance and create a **DataTable** containing the objects we want to insert.

```
using var bulk = new SqlBulkCopy(ConnectionString);

bulk.DestinationTableName = "dbo.Users";

bulk.ColumnMappings.Add(nameof(User.Email), "Email");
bulk.ColumnMappings.Add(nameof(User.FirstName), "FirstName");
bulk.ColumnMappings.Add(nameof(User.LastName), "LastName");
bulk.ColumnMappings.Add(nameof(User.PhoneNumber), "PhoneNumber");
```

```
await bulk.WriteToServerAsync(GetUsersDataTable());
```

However, the performance is blazing fast:

```
SQL Bulk Copy, for 100 users: 1.7 ms
SQL Bulk Copy, for 1,000 users: 7 ms
SQL Bulk Copy, for 10,000 users: 68 ms
SQL Bulk Copy, for 100,000 users: 646 ms
SQL Bulk Copy, for 1,000,000 users: 7,339 ms
```

Here's how you can create a **DataTable** and populate it with a list of objects:

```
DataTable GetUsersDataTable()
{
    var dataTable = new DataTable();

    dataTable.Columns.Add(nameof(User.Email), typeof(string));
    dataTable.Columns.Add(nameof(User.FirstName), typeof(string));
    dataTable.Columns.Add(nameof(User.LastName), typeof(string));
    dataTable.Columns.Add(nameof(User.PhoneNumber), typeof(string));

    foreach (var user in GetUsers())
    {
        dataTable.Rows.Add(
            user.Email, user.FirstName, user.LastName, user.PhoneNumber);
    }

    return dataTable;
}
```

Results

Here are the results for all the bulk insert implementations:

```
| Method             |   Size     |      Speed
|------------------- |----------- |----------------:
| EF_OneByOne        | 100        |       19.80 ms |
| EF_OneByOne        | 1000       |      259.87 ms |
| EF_OneByOne        | 10000      |    8,860.79 ms |
| EF_OneByOne        | 100000     |            N/A |
| EF_OneByOne        | 1000000    |            N/A |

| Dapper_Insert      | 100        |      10.650 ms |
| Dapper_Insert      | 1000       |     113.137 ms |
| Dapper_Insert      | 10000      |   1,027.979 ms |
| Dapper_Insert      | 100000     |  10,916.628 ms |
| Dapper_Insert      | 1000000    | 109,064.815 ms |

| EF_AddAll          | 100        |       2.064 ms |
| EF_AddAll          | 1000       |      17.906 ms |
| EF_AddAll          | 10000      |     202.975 ms |
| EF_AddAll          | 100000     |   2,129.370 ms |
| EF_AddAll          | 1000000    |  21,557.136 ms |

| EF_AddRange        | 100        |       2.035 ms |
| EF_AddRange        | 1000       |      17.857 ms |
| EF_AddRange        | 10000      |     204.029 ms |
| EF_AddRange        | 100000     |   2,111.106 ms |
| EF_AddRange        | 1000000    |  21,605.668 ms |
```

```
| BulkExtensions     | 100        |       1.922 ms |
| BulkExtensions     | 1000       |       7.943 ms |
| BulkExtensions     | 10000      |      76.406 ms |
| BulkExtensions     | 100000     |     742.325 ms |
| BulkExtensions     | 1000000    |   8,333.950 ms |

| BulkCopy           | 100        |       1.721 ms |
| BulkCopy           | 1000       |       7.380 ms |
| BulkCopy           | 10000      |      68.364 ms |
| BulkCopy           | 100000     |     646.219 ms |
| BulkCopy           | 1000000    |   7,339.298 ms |
```

Takeaway

`SqlBulkCopy` holds the crown for maximum raw speed. However, EF Core Bulk Extensions deliver fantastic performance while maintaining the ease of use that Entity Framework Core is known for.

The best choice hinges on your project's specific demands:

- Performance is all that matters? **SqlBulkCopy** is your solution.
- Need excellent speed and streamlined development? EF Core is a smart choice.

I leave it up to you to decide which option is best for your use case.

Hope this was helpful.

See you next week.

Milan Jovanović

Find me online:

X/Twitter / LinkedIn / YouTube

P.S. There are 2 ways I can help you improve your skills:

Modular Monolith Architecture

This in-depth course will transform the way you build monolith systems. You will learn the best practices for applying the Modular Monolith architecture in a real-world scenario.

JOIN THE WAITLIST

Pragmatic Clean Architecture

This is the complete blueprint for building production-ready applications using Clean Architecture. Join **2,500+ other students** to accelerate your growth as a software architect.

START WATCHING NOW

BEFORE YOU LEAVE 👋

How am I doing?

I love hearing from readers, and I'm always looking for feedback. How am I doing with The .NET Weekly? Is there anything you'd like to see more or less of? Which aspects of the newsletter do you enjoy the most?

Hit reply and say hello - I'd love to hear from you!

Stay awesome,

Milan