Final Project Of C# Bootcamp For System Group

Soheil Hajian Manesh

Part1 :

This part of the project is in Part1 project folder and if you run Program , the results are written in Reports folder in .txt format for each query. You should run the program to .txt files in Reports created.
For this part of the project I use a class named Report.cs. The `Report` class is a utility class designed to generate and store various reports mentioned in the part1 of the project. It encapsulates multiple methods, each responsible for executing specific data queries and outputting the results to text files. For each of the queries, I have made assumptions that I will explain below :

Query 1 :

I assume this query wants the number of invoices, sum & average of Total field in all invoices.

Query 2 :

I use *EF.Functions.Random()* to order datas randomly each time i call this method. This function is suitable for the Sqlite Database provided by *Entity Framework Core*.

Query 3 :

I have considered the Quantity * UnitPrice as amount of sales for each invoice line and calculate sum of it for all invoice lines of a track.

Query 4 :

Like previous part I retrieve customers with at least one InvoiceLine with Quantity * UnitPrice < 2

Query 5 :

I have considered the sum of Total fields Of Invoices of a customer as total purchases of a customer.

Query 6 :

Not any specific consideration.

Query 7 :

I considered Total field of an invoice as the amount of sale for that invoice and retrieved the average of that field for all invoices in a month.

Query 8 :

For this query I tried a lot to not using AsEnumerable and do all queries operations in database but i can’t because i use nested select and complex query that if I didn’t use AsEnumerable I get Runtime Error(it says Sqlite does not support APPLY operation) . So I tried to process the query in the database as much as possible and just when it is necessary I use AsEnumerable.

Query 9 :

Not any specific consideration.

Query 10 :

Not any specific consideration.

Part2 :

This part of the project is in the Part2 project folder and if you run Program , tests are run and you can see the result of all tests in the terminal.Also you can test operations manually using TestManually static class .(The code for calling the respective method is commented in program class. You can uncomment if you need.)
For this part I have 3 classes (Filter.cs for Where operation,Project.cs for Select operation,Sort.cs for sort operation) that all of them derived from abstract class DynamicOperator. I use Factory Method Design Pattern for creating and applying each operation.This logic implemented in OperatorsFactory.cs.
I have created some static classes for unit testing and also a static class TestManually.cs that you can use in order to test each operator using a terminal.
In the design of dynamic projection I create a new instance of the same class that I want to project on it and all fields that I do not want to select, filled with null or 0 in that instance. Of course in reality it’s most of the time not to work this way and we construct an anonymous type using new {}, but implementing this using *Expression* was a little hard so after some attempts I decided to go with an easier way I explained.

Part3 :

This part of the project is designed in the “FinalProject” project folder and if you run Program , the results are written in Reports folder in .txt format for each part. In the main program both methods are running .(sequential & parallel loading) You should run the program to .txt files in Reports created.

Plugin Development:

For implementing the plugin architecture I have three class libraries.First one ,Plugin.Common that has ILoyaltyClubPoint interface.Second one, Plugin.MonthlyPurchasedPoint that implements ILoyaltyClubPoint interface and for calculate loyalty points based on the number of tracks purchased per month.Third one, Plugin.SeasonalBonus that implements ILoyaltyClubPoint as well and for calculate loyalty club point based on seasonal bonus.
In both methods in the 2nd & 3rd classes I retrieve a Dictionary that have these key value pairs : (customer Id,total loyalty point earned)
In the Final Project I have implemented two classes in order to use plugins. First one,PluginLoadContext that derived from AssemblyLoadContext and it’s duty is to load plugins assemblies from their path and also resolve dependencies.This class almost similar to what was designed in the book and what we did in the class.I also design a static handler class that used to handle Loyalty Club.Load and execute each plugins and add points of each customer to a Dictionary<customer,point> .Finally I wrote total Loyalty point of each customer in the .txt file.

Parallel Loading:

For this part I only define the CalculateLoyaltyClubPointsAsync method that is different from *CalculateLoyaltyClubPoints* that used in the previous part. On that method I load and execute plugins sequentially. First MonthlyPurchasedPoint plugin and after its execution finished load and execute SeasonalBonus plugin.but in this method load & execution of plugins are asynchronous.
Two tasks are created and run in parallel:

- `task1` loads and executes a plugin from the `MonthlyPurchasedPointPluginPath` using the existing `dbContext`.

    - `task2` loads and executes a plugin from the `SeasonalBonusPluginPath` using a new instance of `AppDbContext`.

The method waits for both tasks to complete using `Task.WhenAll`.
After both tasks complete, the method writes the loyalty points to a different file from the previous part.
Because this scenario where two threads try to change *_loyaltyClubPoint* at the same time could happen,I use lock for *AddPointsToClub* to avoid race conditions.

- **More about parallel loading of plug-ins**

To enhance the performance and responsiveness of the application, especially when multiple plugins are being loaded and executed, we implemented parallel loading using .NET's `System.Threading.Tasks` namespace. This namespace provides a high-level interface for asynchronously executing callables using tasks.

Implementation

1. **Executor Setup**: We use `Task` for managing asynchronous execution of plugin loading and calculation tasks concurrently.
2. **Loading Plugins in Parallel:** When the application starts, it initializes the plugins in parallel, ensuring that the main thread is not blocked during this process.
3. **Executing Plugin Methods Concurrently**: During the loyalty points calculation, each plugin's `CalculatePoints` method is executed concurrently to further enhance performance.

In our approach we merge steps 2 & 3.

Benefits of Parallel Loading

1. **Improved Performance**: By loading and executing plugins concurrently, the overall time required for these operations is reduced, leading to faster application response times.

2. **Scalability**: As the number of plugins grows, parallel loading ensures that the system can handle the increased load without significant degradation in performance.
3. **Responsiveness**: Users experience a more responsive application, as the main thread remains unblocked and available for other tasks.

The implementation of parallel loading for plugins in our loyalty club points calculation system leverages .NET's `System.Threading.Tasks` namespace to achieve significant performance improvements. By executing plugin loading and calculations concurrently, we ensure that the application remains responsive and scalable, capable of handling multiple plugins efficiently.