Spurs Ltd System Report

Analysis and Design.

Use of Task Parallel Library

Student:

Ezequiel Farioli

Teacher:

Davin Ward – Craig Weightman

Course:

Task-Based Software Engineering

**Analysis and Design**

The approach I have adopted to build this system was to focus on the main two problems: how to load the data as fast as possible, and once the data is load in appropriates data structures, how to query the data as well rapidly.

I knew that I could use the Task Parallel Library TPL to run tasks and processes in parallel to make more efficient use of the processor of the computer.

So, using Parallel loops, I seem to be the right answer to try to optimize the loading process. I try to nest two Parallel loops, but this creates an overhead, due to the more chances of deadlocks. So, after several testing, I concluded that using the parallelism in the outer loop increases the efficiency of the loading process.

**Load Data.**

Due to while I am loading the data in data-structure that allow organizing the information, and this load is done by many treads, I need a data structure that is a thread-safe. So in the first instance, I thought of using Concurrent-Bag, which is a list but thread-safe without the possibility of accessing by index. However, I looked the way of CVS files were organized, and I deducted that I could use a concurrent dictionary, using the name of the files as the index compose by the store id, the year and the week, and its value will be the whole list of rows which comes as an array of string, but I convert this as Order detail (see Class Diagram in appendix). So I have a part of code that creates an Index object from the name of the file, and another method that transforms the array of the string that I have from read-all-lines into an array of Order Detail, which may worth to point it out that is the inner loop that cycles sequentially.

**Method for a General Query.**

To resolve the part of the query I have decided to create a "master query” which will do any query required according to the parameters passed will return the sum of all elements matched in such query. If I pass a store code, the query will match all the elements with that store code and return the sum of all of them. The query can match using five parameters that can be used at the same time. The way to avoid using a parameter as part of the query is passing null when is a parameter of type string or zero ("0") if it is an integer.

The query maximizes the use of parallelism to find the sum. In a part of the algorithm, I must iterate the concurrent dictionary, and I do in a parallel way, so I have to store the sum in a concurrent bag of decimal which I sum at the end.

Another problem that I had was how to make a parameter have a wildcard when it is passed null or 0, and the solution I have was, through another method I did, when the parameter passed was null or 0 the where clause equal to itself. (please see how it is implemented “FirstOrDefault” method in the “querytotals” method).

Worth to mention, I thought to implement the builder pattern with the query total methods, so instead of passing the 5 parameters, I just passed what I need or even none parameter if I want to get the big total for the whole franchise. But I think It did not worth the effort, besides to do this I should make “querytotals” its own class.

**Creating and Populating HashSet.**

After the load of the orders and stores, I created 3 HashSet, with dates, supplier names and supplier types. This HashSet helps me to populate my combo box in my GUI app, and for my console app in F#, the hash sets are the piece of information which I used to validate all my inputs.

Worth to mention that for the F# app for the validation of the dates I have created a HashSet of Dates, However for validating input for year and week, I am using a Dictionary that created from the HashSet of Dates. The key of the dictionary in the year, and for each year I have a HashSet of int for the weeks. Well, they technically are a dictionary of strings with a HashSet of string, but those strings are integer values in a string format as in "2014" or "3", etc.

Entity Relationship Diagram.

Although the program does not load the data in any database, in my opinion, it helps to visualize how the records will be load in memory by representing them in an Entity Relationship Diagram or ERD, please the appendix for the corresponding diagram.

**GUI implementation with windows forms.**

To complete this part as I say before I use Hash-Sets that like second part of the load.

In each for there is text-label which have a yellow background that denotes the totals asked in the first nine uses cases (see uses cases diagram). The other use cases are completed with the incorporation of combo-box that shows Stores, year, week, supplier name and supplier type as it is needed, depends on which screen you are.

**I have added some extra features which considered can be useful, but like add I have no background in finances or economic, so it is hard to tell for me if they are helpful or not. However, I believe that I can quickly implement any feature that the Spurs Ltd Finance Team could need and I to its entire predisposition.**

I can name some of the features:

* For example, for the use case 4, show three charts. The first show the whole year divide in weeks, if you hover over an individual bar will show the week number and the total of sales.

The second and third graphics are related to the selected week in the combo-box which total is displayed in background yellow, as I mentioned earlier.

The second chart compares a particular week with other weeks for the same Store. It will show in which week of the year the store sold the least, the average for the whole year, the currently selected week, and in which week the store sold the most.

The third chart will compare that week with the sales of that week for all the stores. The first column will be for the store has sold least that week; the second column will be for the average of sales for stores that year: the third column will be for the currently selected store, and the last column will be for the store which has sold the most that week of the year.

As a visual aid, the week chosen is coloured orange in the three graphics.

* In use case 8, it is similar to the first graphics of use case 4, but It has five checkboxes.

The first will Highlight the column corresponding to the selected week with an outline orange.

The second will show the average red line for the average of sell for that store and supplier type that year.

The third checkbox will highlight the column for the three weeks with least sells, in case they are not zero, they will be painted red.

Similarly, the fourth check-box will highlight the column for the three weeks with most sells; they will be painted green.

Lastly, the last checkbox will show another series, in the form of points with background green and outline black. The information in each point will be the average that all the stores have done in that week.

All other case has some graphics that pretend to aim in a visual and intuitive way to help make most of the information.

**Application Performance Analysis.**

I have tested three versions of the application in two different computers.

* The first version of the application was built in C# without using the Parallel library.
* The second used the Parallel library.
* The last version was built in F# taking advantage of the parallelism.

The computers used for this test were:

* AMD Ryzen 7 2700X, 8 CPU cores with 16 threads at 3.1 GHz.
* Intel i5 4440, 4 CPU cores with 4 threads at 3.1 GHz

The main parts of the application that have been tested are the load of the Stores and the Orders into two suitable data structures, and getting the total of a specific Supplier, which is undoubtedly the most ponderous process that the system performs.

These tasks have run fifty times and I have extracted the average and express the result in milliseconds in the following table:

|  |  |  |
| --- | --- | --- |
|  | AMD Ryzen 7 2700X | Intel i5 4440 |
| Sequential C# Load | 5683.09 | 7967.17 |
| Parallel C# Load | 3740.50 | 5500.57 |
| Parallel F# Load | 4766.89 | 6515.10 |
| Sequential C# Query | 789.05 | 1014.96 |
| Parallel C# Query | 89.99 | 102.44 |
| Parallel F# Query | 62.13 | 132.46 |

The sequential load takes a 52% more time to load than a C# parallel load and up to 19% more than an F# parallel load; in the AMD processor.

The sequential load takes a 45% more time to load than a C# parallel load and up to 22% more than an F# parallel load; in the Intel processor.

The sequential query takes 877% more time to load than a C# parallel load and up to 1270% more than an F# parallel load; in the AMD processor.

The sequential query takes 991% more time to load than a C# parallel load and up to 766% more than an F# parallel load; in the Intel processor.

















  

