

Przemysłowe Bazy Danych

*Aplikacja testująca szybkość zapisu do plików tekstowych oraz
bazy danych MS SQL Server*

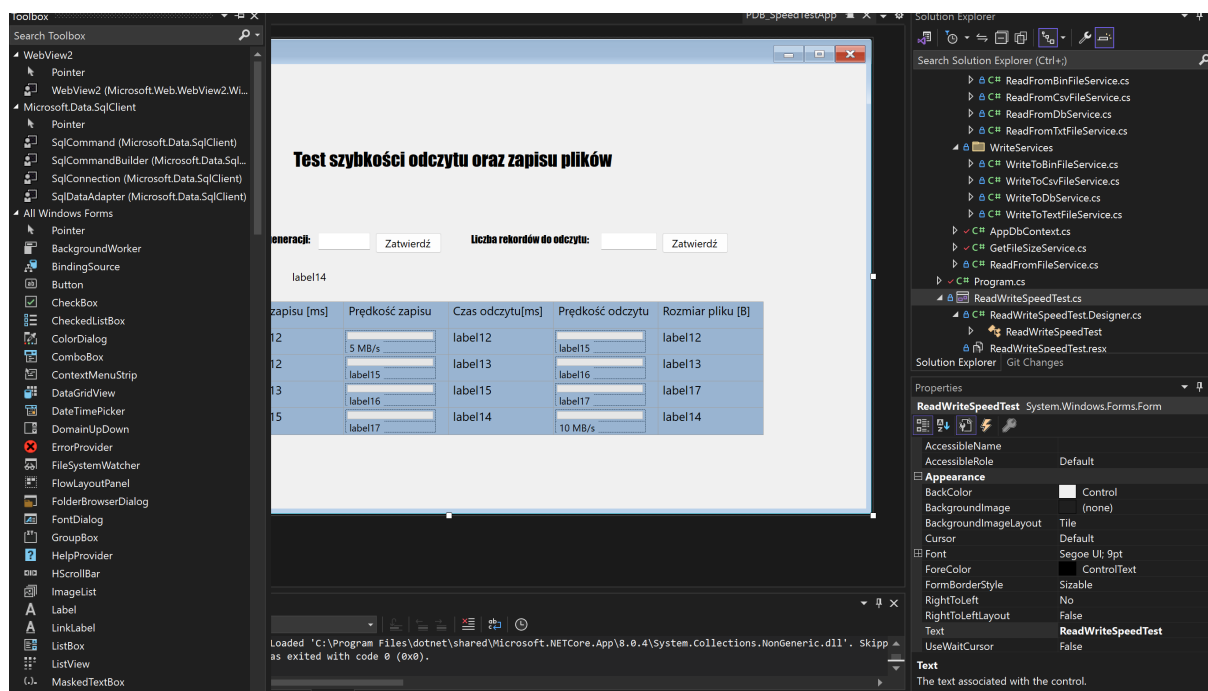
Autorzy:

Jakub Pająk
Kacper Stiborski

AiR Grupa 5TI

Spis treści

1.	Wprowadzenie	2
1.1.	Cel projektu	2
2.	Realizacja projektu	2
2.1.	Interfejs użytkownika	2
2.2.	Obsługa zdarzeń i logika aplikacji	2
2.3.	Implementacja metod zapisu do pliku	3
2.4.	Implementacja zapisu do bazy danych	5
2.5.	Implementacja metod odczytu z pliku	6
2.6.	Implementacja odczytu danych z bazy danych	8
3.	Działanie aplikacji	9
3.1.	Generacja pliku wsadowego oraz jego zapis	9
3.2.	Odczyt pliku	9
3.3.	Wizualizacja danych	9
3.4.	Zabezpieczenia	9
4.	Wnioski	12



Rysunek 1: Widok projektanta Windows Forms w IDE Visual Studio 2022

1. Wprowadzenie

1.1. Cel projektu

Celem projektu była implementacja aplikacji desktopowej, której zadaniem było wykonywanie operacji na plikach tekstowych oraz bazie danych. Aplikacja miała umożliwiać zapis określonej przez użytkownika liczby rekordów do pliku tekstowego oraz ich odczyt, a także dokonywać pomiaru czasu niezbędnego do wykonania tych operacji. Dodatkowo, projekt zakładał realizację analogicznych operacji na bazie danych MS SQL Server, z uwzględnieniem zapisu i odczytu rekordów oraz pomiaru wydajności obu procesów. Celem porównania efektywności operacji na plikach oraz bazie danych, aplikacja miała dostarczyć użytkownikowi czytelne informacje o czasie trwania poszczególnych zadań.

2. Realizacja projektu

2.1. Interfejs użytkownika

Interfejs użytkownika został zaprojektowany przy użyciu wbudowanego projektanta graficznego, będącego częścią frameworka *Windows Forms*. Proces ten polegał na wybieraniu z przybornika (ang. *Toolbox*) odpowiednich elementów funkcjonalnych, takich jak przyciski, etykiety czy pola tekstowe, i umieszczaniu ich na obszarze roboczym okna aplikacji. Dzięki narzędziu *Properties*, możliwe było szczegółowe dostosowanie wyglądu i zachowania poszczególnych kontroltek do specyficznych wymagań projektowych, takich jak rozmiar, położenie, czcionka, czy reakcje na zdarzenia użytkownika.

2.2. Obsługa zdarzeń i logika aplikacji

Funkcja odpowiedzialna za inicjalizację oraz obsługę zdarzeń na głównym widoku aplikacji ma prostą, lecz efektywną strukturę. W konstruktorze następuje inicjalizacja widoku oraz przypisanie wartości początkowych zmiennym niezbędnym do działania aplikacji. Wszystkie kolejne funkcje związane z obsługą zdarzeń są generowane automatycznie przez Visual Studio, na przykład poprzez dwukrotne kliknięcie na kontrolkę w widoku projektanta. Tak właśnie zrealizowano obsługę przycisków w opisywanej aplikacji.

Główna funkcja logiki aplikacji wywołuje dwie metody pomocnicze: *InvokeWriteServices()* oraz *InvokeReadServices()*. Metody te z kolei korzystają z odpowiednich klas, które obsługują zapis i odczyt danych,

```

1 private void submitWrite_Click(object sender, EventArgs e)
2 {
3     int amount = textBox_inputAmount.Text.Length > 0 ? int.Parse(textBox_inputAmount.Text) : 0;
4
5
6     // zabezpieczenie i return
7     if (amount <= 0.0)
8     {
9         lbl_InputWarning.Text = "UWAGA! Proszę wprowadzić poprawną ilość danych do generacji (n > 0).";
10        lbl_InputWarning.ForeColor = System.Drawing.Color.Red;
11        return;
12    }
13
14    // nadpisz zmienna do odczytu
15    textBox_outputAmount.Text = amount.ToString();
16
17    var writeHelper = new InvokeWriteServices(_context, amount);
18    Dictionary<string, double> elapsedTimeForFiles_Write = writeHelper.InvokeReadServices();
19    int[] speed = new int[4];
20    //var readHelper = new InvokeReadServicesHelper(_context);
21    //Dictionary<string, double> elapsedTimeForFiles_Read = readHelper.InvokeReadServices();
22
23    GetFileSizeService getFileSizeService = new GetFileSizeService();
24    var sizes = getFileSizeService.GetFileSize();
25
26
27
28    writtenAmount = amount;
29
30    lbl_TimeBin.Text = elapsedTimeForFiles_Write["bin"].ToString();
31    lbl_TimeCSV.Text = elapsedTimeForFiles_Write["csv"].ToString();
32    lbl_TimeTxt.Text = elapsedTimeForFiles_Write["txt"].ToString();
33    lbl_TimeSQL.Text = elapsedTimeForFiles_Write["sql"].ToString();
34

```

Rysunek 2: Fragment kodu przedstawiający implementację obsługi zdarzeń na interfejsie użytkownika

zwracając wyniki w postaci zmiennej typu *Dictionary<string, double>*. Klucz typu *string* odpowiada za identyfikację typu pliku, natomiast wartość typu *double* przechowuje czas potrzebny na wykonanie operacji.

Następnie, czasy operacji są przypisywane do obiektów typu *Label*, co pozwala na ich wyświetlenie w interfejsie użytkownika, dostarczając czytelne informacje o wydajności działania aplikacji.

2.3. Implementacja metod zapisu do pliku

Wszystkie metody odpowiedzialne za zapis do pliku tekstowego są do siebie bardzo podobne. Pierwszym krokiem jest wybranie odpowiedniej ścieżki, w której plik zostanie zapisany. W języku C# można wykorzystać klasę *Environment*, co umożliwia tworzenie dynamicznych ścieżek zależnych od systemu operacyjnego użytkownika.

Kolejny etap to wygenerowanie odpowiedniej ilości rekordów. Dane są generowane za pomocą biblioteki *Bogus*, która pozwala na generowanie realistycznie wyglądających danych testowych, co znacząco ułatwia testowanie aplikacji.

Zasadnicza różnica między klasami obsługującymi zapis do plików różnych typów dotyczy jedynie metody samego zapisu. Dla plików binarnych używana jest klasa *BinaryWriter*, natomiast dla plików tekstowych i CSV stosuje się klasę *StreamWriter*. Dzięki temu każda metoda zapisu jest dostosowana do specyfiki odpowiedniego formatu pliku.

```

1  internal class WriteToBinFileService
2  {
3      public WriteToBinFileService()
4      {
5      }
6
7      public double WriteToBinaryFile(int amount)
8      {
9          string desktopPath = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
10         string folderPath = Path.Combine(desktopPath, "PDB");
11         string filePath = Path.Combine(folderPath, "BinFileTest.bin");
12
13         if (!Directory.Exists(folderPath))
14         {
15             Directory.CreateDirectory(folderPath);
16         }
17
18         GenerateDummyDataService generateDummyData = new GenerateDummyDataService();
19
20         Stopwatch sw = new Stopwatch();
21         double elapsedTime = 0;
22         List<BasicDataDto> data = generateDummyData.GenerateDummyData(amount);
23
24         using (FileStream fs = new FileStream(filePath, FileMode.Create, FileAccess.Write, FileShare.None))
25         using (BinaryWriter writer = new BinaryWriter(fs))
26         {
27             sw.Start();
28             foreach (var item in data)
29             {
30                 WriteBinary(writer, item);
31             }
32             sw.Stop();
33         }
34
35         elapsedTime = sw.Elapsed.TotalMilliseconds;
36         return elapsedTime;
37     }
38     private void WriteBinary(BinaryWriter writer, BasicDataDto item)
39     {
40         writer.Write(item.Name);
41         writer.Write(item.Surname);
42         writer.Write(item.DateOfBirth.ToString());
43         writer.Write(item.Phone);
44     }
45 }

```

Rysunek 3: Klasa realizująca zapis do pliku w formacie binarnym

2.4. Implementacja zapisu do bazy danych

Proces zapisu do bazy danych wymagał zainstalowania odpowiednich paczek, aby umożliwić korzystanie z *Entity Framework*. Wymagane paczki to:

- *Microsoft.EntityFrameworkCore*
- *Microsoft.EntityFrameworkCore.SqlServer*
- *Microsoft.EntityFrameworkCore.Tools*

Dodatkowo, do przeprowadzenia migracji, czyli sposobu, w jaki *Entity Framework* zarządza zmianami w strukturze bazy danych, konieczne było upewnienie się, że pakiet *.NET CLI* jest zainstalowany.

Pierwszym krokiem integracji aplikacji z bazą danych było stworzenie modelu danych. W omawianej aplikacji model ten jest reprezentowany przez klasę *BasicDataDto.cs*, która odpowiada encji bazodanowej. Następnie, w celu skonfigurowania modelu, konieczne było nadpisanie metody *OnModelCreating()* w klasie dziedziczącej po *DbContext*. Taki zabieg informuje *Entity Framework*, że dana klasa odpowiada za konfigurację struktury bazy danych.



```
1  protected override void OnModelCreating(ModelBuilder modelBuilder)
2  {
3      modelBuilder.Entity<BasicDataDto>(entity =>
4      {
5          entity.HasKey(e => e.Id);
6
7          entity.Property(e => e.Name)
8              .IsRequired();
9
10         entity.Property(e => e.Surname)
11             .IsRequired();
12
13         entity.Property(e => e.Phone)
14             .IsRequired();
15     });
16
17     base.OnModelCreating(modelBuilder);
18 }
```

Rysunek 4: Metoda dziedziczona po klasie *DbContext*, nadpisywana w klasie *AppDbContext*, konfigurująca encje bazy danych

Aby aplikacja mogła komunikować się z bazą danych, musiała zostać uruchomiona wewnątrz klauzuli *using* w funkcji *Main()*, co zapewnia odpowiednie zarządzanie połączeniami z bazą podczas działania programu.

```

1 internal static class Program
2 {
3     /// <summary>
4     /// The main entry point for the application.
5     /// </summary>
6     [STAThread]
7     static void Main()
8     {
9         // To customize application configuration such as set high DPI settings or default font,
10        // see https://aka.ms/applicationconfiguration.
11
12        var optionsBuilder = new DbContextOptionsBuilder<AppDbContext>();
13        optionsBuilder.UseSqlServer(@"Server=jakubpajak_asus;Database=pdb_database;Trusted_Connection=True;TrustServerCertificate=True;");
14        //optionsBuilder.UseSqlServer(@"Server=KACPER\SQLEXPRESS;Database=PDB;Trusted_Connection=True;TrustServerCertificate=True;");
15        var dbContextOptions = optionsBuilder.Options;
16
17        ApplicationConfiguration.Initialize();
18
19        using (var context = new AppDbContext(dbContextOptions))
20        {
21            context.Database.Migrate();
22            Application.Run(new ReadWriteSpeedTest(context));
23        }
24    }
25 }

```

Rysunek 5: Klasa będąca punktem startowym aplikacji. Znajduje się w niej uruchomienie aplikacji oraz wywołanie metody aktualizującej migracje

Sam proces zapisu danych do bazy jest stosunkowo prosty. Do kontekstu bazy danych dodawane są nowe rekordy, po czym metoda *SaveChanges()* zapisuje zmiany w bazie danych, zapewniając trwałość operacji.

```

1 public double WriteToDatabase(int amount)
2 {
3     Stopwatch sw = new Stopwatch();
4     GenerateDummyDataService generateDummyDataService = new GenerateDummyDataService();
5
6     List<BasicDataDto> basicDataDtos = generateDummyDataService.GenerateDummyData(amount);
7
8     sw.Start();
9     _appDbContext.AddRange(basicDataDtos);
10    _appDbContext.SaveChanges();
11    sw.Stop();
12
13    return sw.Elapsed.TotalMilliseconds;
14 }

```

Rysunek 6: Metoda odpowiadająca za zapis rekordów do bazy danych

2.5. Implementacja metod odczytu z pliku

Implementacja metod odczytu z pliku we wszystkich metodach przebiega bardzo podobnie. Pierwszym etapem jest utworzenie ścieżki do pliku za pomocą klasy *Environment*. Następnie tworzona jest instancja klasy *Stopwatch* w celu pomiaru czasu potrzebnego do odczytania danych. Kolejnym krokiem jest właściwy odczyt z pliku, realizowany wewnątrz struktury *try-catch*. W przypadku odczytu z pliku binarnego skorzystano z klasy *BinaryReader*. W pozostałych klasach odczytujących z pliku tekstowego oraz CSV, wykorzystano klasę *StreamReader*.

```

1 public double ReadFromBinFile(int amount)
2 {
3     List<string> retrievedData = new List<string>();
4
5     string desktopPath = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
6     string folderPath = Path.Combine(desktopPath, "PDB");
7     string filePath = Path.Combine(folderPath, "BinFileTest.bin");
8
9     Stopwatch sw = new Stopwatch();
10
11     try
12     {
13         using (FileStream fs = new FileStream(filePath, FileMode.Open, FileAccess.Read))
14         using (BinaryReader reader = new BinaryReader(fs))
15         {
16             sw.Start();
17             //while (fs.Position < fs.Length)
18             for (int i = 0; i < amount; i++)
19             {
20                 string name = reader.ReadString();
21                 string surname = reader.ReadString();
22                 string dateOfBirth = reader.ReadString();
23                 string phone = reader.ReadString();
24
25                 retrievedData.Add($"{name},{surname},{dateOfBirth},{phone}");
26             }
27             sw.Stop();
28         }
29     }
30     catch (Exception ex)
31     {
32         Console.WriteLine($"An error occurred: {ex.Message}");
33         return -1;
34     }
35
36     return sw.Elapsed.TotalMilliseconds;
37 }

```

Rysunek 7: Metoda odpowiadająca za odczyt rekordów z pliku binarnego


```

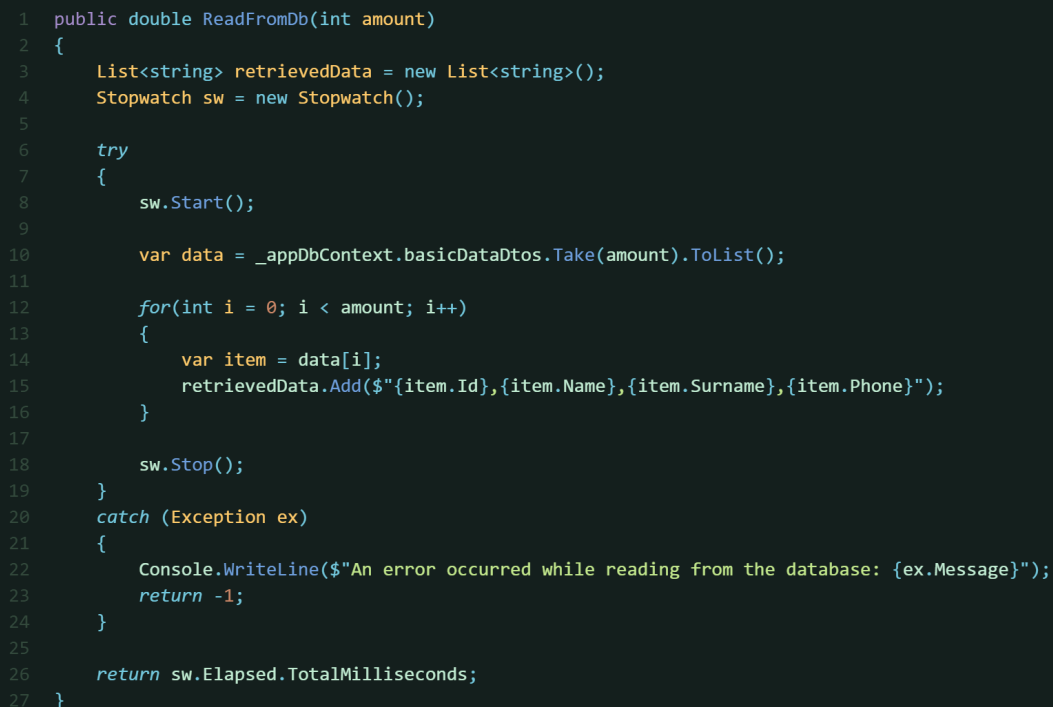
1 public double ReadFromCsvFile(int amount)
2 {
3     List<string> retrievedData = new List<string>();
4
5     string desktopPath = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
6     string folderPath = Path.Combine(desktopPath, "PDB");
7     string filePath = Path.Combine(folderPath, "CsvFileTest.csv");
8
9     Stopwatch sw = new Stopwatch();
10
11     try
12     {
13         using (StreamReader streamReader = new StreamReader(filePath))
14         {
15             sw.Start();
16             //while (!streamReader.EndOfStream)
17             for(int i = 0; i < amount; i++)
18             {
19                 retrievedData.Add(streamReader.ReadLine());
20             }
21             sw.Stop();
22         }
23     }
24     catch (Exception ex)
25     {
26         Console.WriteLine($"An error occurred: {ex.Message}");
27         return -1;
28     }
29
30     return sw.Elapsed.TotalMilliseconds;
31 }

```

Rysunek 8: Metoda odpowiadająca za odczyt rekordów z pliku CSV

2.6. Implementacja odczytu danych z bazy danych

Implementacja odczytu rekordów z bazy danych jest prosta - opiera się na wykorzystaniu metody *Take()* wywołanej na obiekcie kontekstu bazy danych. Jednocześnie uwidacznia to wielką zaletę korzystania z EF - programista nie musi korzystać bezpośrednio z zapytań SQL, tylko wprost z języka C# oraz wyrażeń *lambda* i metod udostępnionych na obiekcie *DbContext*.



```

1 public double ReadFromDb(int amount)
2 {
3     List<string> retrievedData = new List<string>();
4     Stopwatch sw = new Stopwatch();
5
6     try
7     {
8         sw.Start();
9
10        var data = _appDbContext.basicDataDtos.Take(amount).ToList();
11
12        for(int i = 0; i < amount; i++)
13        {
14            var item = data[i];
15            retrievedData.Add($"{item.Id},{item.Name},{item.Surname},{item.Phone}");
16        }
17
18        sw.Stop();
19    }
20    catch (Exception ex)
21    {
22        Console.WriteLine($"An error occurred while reading from the database: {ex.Message}");
23        return -1;
24    }
25
26    return sw.Elapsed.TotalMilliseconds;
27 }

```

Rysunek 9: Metoda odpowiadająca za odczyt rekordów z bazy danych

3. Działanie aplikacji

3.1. Generacja pliku wsadowego oraz jego zapis

Użytkownik po uruchomieniu aplikacji może wygenerować plik o określonej przez niego długości rekordów a następnie dokoknuje się zapis pliku w formacie .txt, .bin, .csv oraz bazodanowy.

3.2. Odczyt pliku

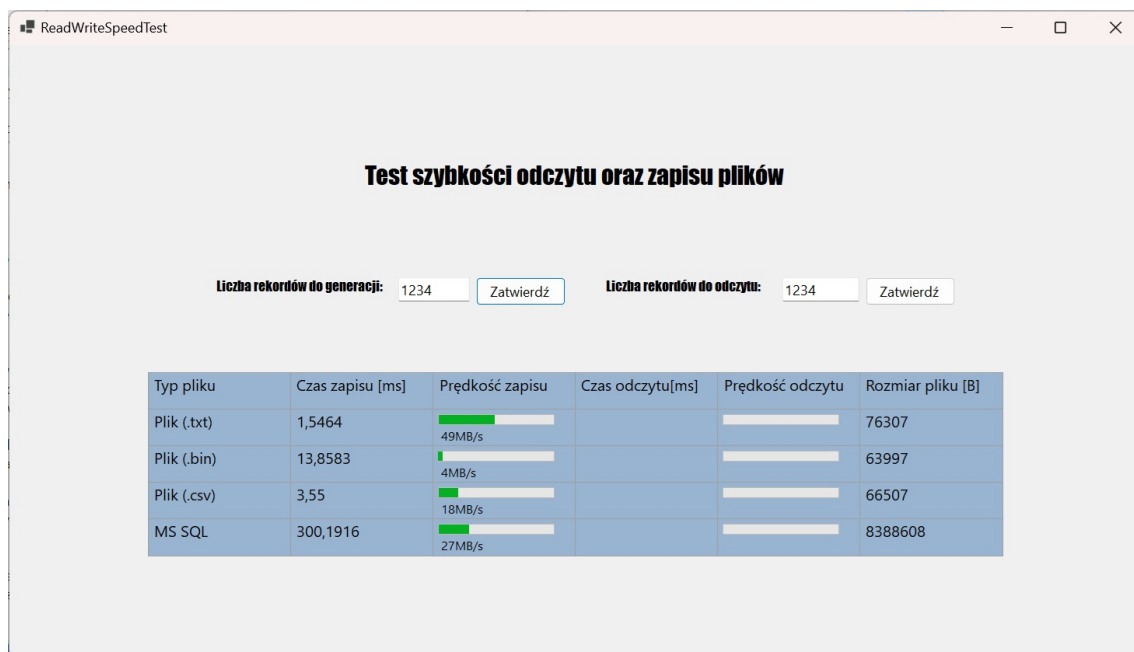
Po zapisaniu plików można przejść do kolejnego okienka, w którym należy podać ilość rekordów z pliku do odczytu. Automatycznie przepisywana jest wartość określona przy zapisie lecz użytkownik może dowolnie zmienić tą ilość.

3.3. Wizualizacja danych

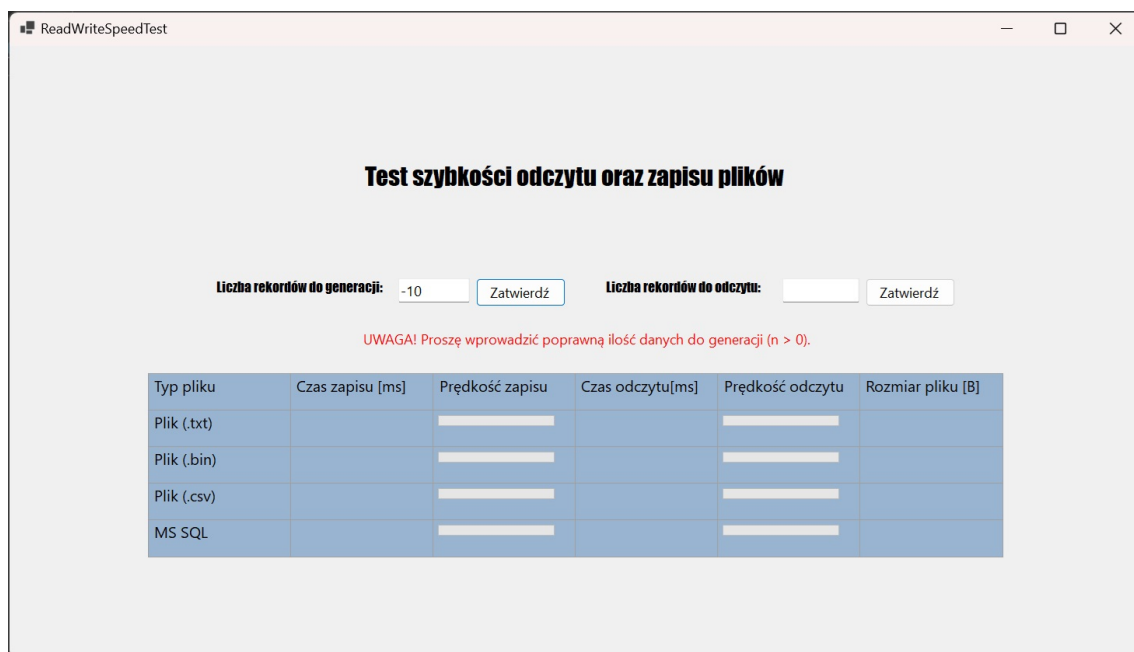
W tabeli poniżej okienek do wprowadzania wartości po zapisie i odczycie aktualizowana jest tabela z właściwościami: czasu zapisu i odczytu w milisekundach, prędkości zapisu i odczytu w MB/s oraz rozmiarem pliku w bajtach. Dodatkowo paski obrazujące prędkość zapisu skalują się wraz z rosnącą wartością prędkości, aby uniknąć błędów programu.

3.4. Zabezpieczenia

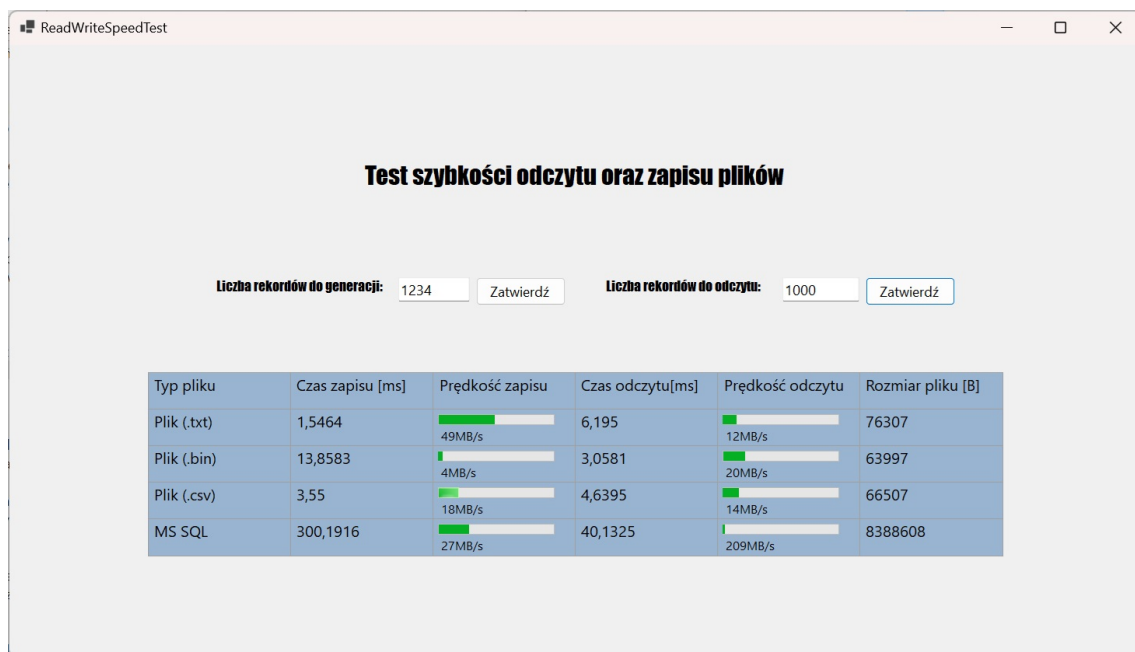
Aplikacja posiada zabezpieczenia przed wpisaniem ujemnej wartości rekordów oraz przed wybraniem do odczytu większej ilości rekordów niż zostały zapisane, co mogłoby powodować poważne usterki programu. Jak w akapicie wyżej, zastosowano także zabezpieczenie przed przepełnieniem wartości wizualizowanej przez paski postępu.



Rysunek 10: Zapis plików



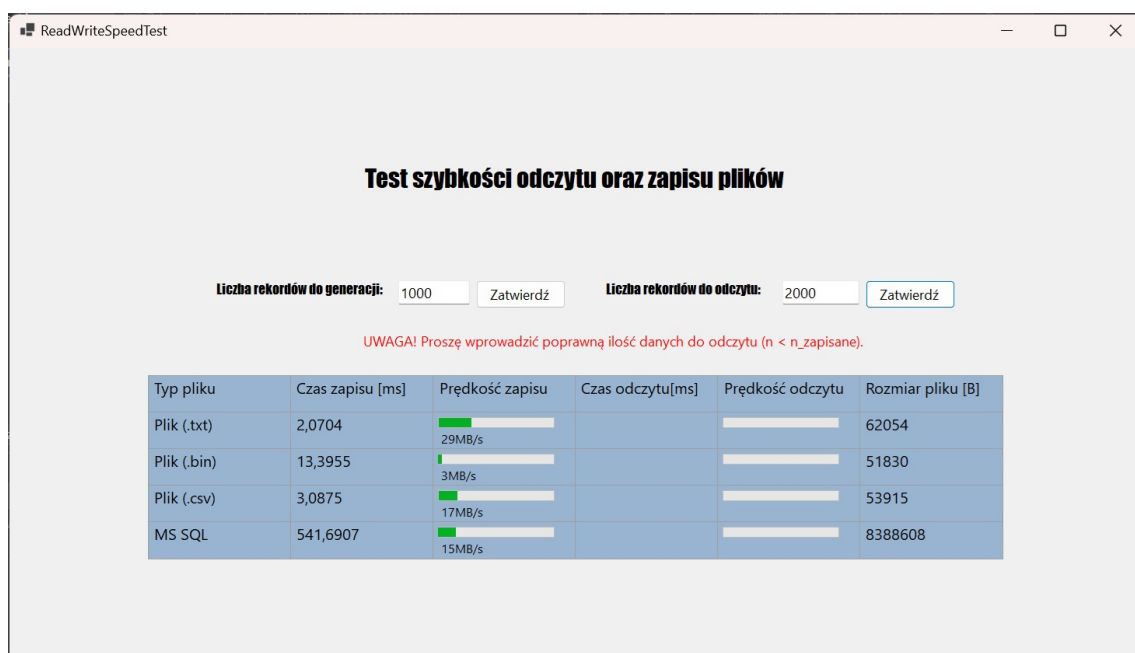
Rysunek 13: Błąd zapisu



Rysunek 11: Odczyt plików

Typ pliku	Czas zapisu [ms]	Prędkość zapisu	Czas odczytu[ms]	Prędkość odczytu	Rozmiar pliku [B]
Plik (.txt)	5,4037	57MB/s	17,6009	17MB/s	309417
Plik (.bin)	2,8253	91MB/s	11,3814	22MB/s	259292
Plik (.csv)	2,8495	94MB/s	15,9284	16MB/s	269379
MS SQL	667,5636	113MB/s	43,4402	1737MB/s	75497472

Rysunek 12: Tabela danych



Rysunek 14: Błąd odczytu

4. Wnioski

Na podstawie wyników uzyskanych podczas testów szybkości zapisu danych do różnych typów plików oraz bazy danych stwierdzono, że zapis do bazy danych wymaga najwięcej czasu. Najszybszy okazał się zapis do pliku binarnego, jednak różnice między nim a plikiem tekstowym oraz CSV są niewielkie i wynoszą około 2 ms przy dużych zbiorach danych. Wraz ze zmniejszaniem się liczby danych różnice te stają się mniej istotne.

Odczyt danych wykazał podobne zależności. Istnieje bardziej wyraźna różnica między czasem odczytu z pliku binarnego a innymi typami plików, jednak odczyt z bazy danych okazuje się szybszy, co częściowo zmniejsza tę różnicę. Niemniej jednak, baza danych nadal charakteryzuje się istotnie dłuższym czasem operacji w porównaniu z pozostałymi metodami.

Liczba rekordów	Typ pliku	Czas zapisu [ms]	Prędkość zapisu [MB/s]	Czas odczytu [ms]	Prędkość odczytu [MB/s]	Rozmiar pliku [B]
10	.txt	0,0165	36	0,8095	0	603
10	.bin	0,0293	17	0,3432	1	503
10	.csv	0,015	38	0,6136	0	574
10	MS SQL	37,5979	2008	33,9804	2221	75497472
100	.txt	0,2252	27	0,8095	6	6162
100	.bin	0,2181	23	0,3432	9	5150
100	.csv	0,2027	26	0,6136	7	5429
100	MS SQL	78,5148	961	33,9804	3362	75497472
1000	.txt	0,7274	85	2,1942	28	61900
1000	.bin	1,0349	50	1,3244	39	52144
1000	.csv	0,6735	79	1,7552	30	53851
1000	MS SQL	164,997	457	33,5489	2250	75497472
5000	.txt	2,9831	103	11,1094	27	309149
5000	.bin	2,7889	93	5,796	44	259502
5000	.csv	3,0018	89	9,3305	28	269393
5000	MS SQL	700,947	107	53,8153	1402	75497472
10000	.txt	6,6558	92	31,5292	19	618154
10000	.bin	6,2869	82	24,517	21	519163
10000	.csv	6,2394	86	27,86	19	538538
10000	MS SQL	1288,6288	58	151,4981	498	75497472
20000	.txt	12,1105	102	49,1183	25	1238614
20000	.bin	10,5348	98	27,5653	37	1037164
20000	.csv	11,6446	92	42,3559	25	1078032
20000	MS SQL	2597,093	29	274,9041	274	75497472
50000	.txt	75,8124	40	49,1183	25	3094244
50000	.bin	48,1182	53	27,5653	37	2592344
50000	.csv	47,3513	56	42,3559	25	2692995
50000	MS SQL	8626,9349	8	274,9041	274	75497472
100000	.txt	125,742	49	225,8607	27	6188232
100000	.bin	168,2461	30	132,9774	39	5187281
100000	.csv	75,4342	71	165,8085	32	5388464
100000	MS SQL	17074,6049	4	1093,335	69	75497472

Rysunek 15: Porównanie