Gradus Vol 10, No 2 (2023)
ISSN 2064-8014

HOGYAN ÉRDEMES NAGY TÖMEGŰ ADATOT IMPORTÁLNI MICROSOFT .NET FRAMEWORK PLATFORMON?

HOW IS IT RECOMMENDED TO IMPORT LARGE AMOUNTS OF DATA USING THE MICROSOFT .NET FRAMEWORK PLATFORM?

Kaczur Sándor[0009-0009-0118-2575 1*], *Friedel Attila*[0009-0003-2528-4302 1]

[1] it-tanfolyam.hu, Magyarország
https://doi.org/10.47833/2023.2.CSC.006

Kulcsszavak:
oktatás
it-tanfolyam.hu
adatbázis-kezelés
Microsoft .NET Framework
adatok importálása

Keywords:
education
it-tanfolyam.hu
database management
Microsoft .NET Framework
data import

Cikktörténet:
Beérkezett 2023. augusztus 25.
Átdolgozva 2023. október 25.
Elfogadva 2023. október 30.

Összefoglalás

Üzleti alkalmazások fejlesztésénél elengedhetetlen alkotóelem az adatok kezelése, tárolása. Ezt leggyakrabban valamilyen relációs adatbázis-kezelővel valósítják meg a fejlesztők. A hétköznapi munka során gyakran előforduló feladat külső forrásból történő adatok átvétele, aktualizálása. A cikk szerzői arra a kérdésre keresik a választ, hogy hogyan érdemes ezen (néha igen tetemes mennyiségű) adatokat minél gyorsabban átvenni. A bemutatásra kerülő esettanulmány Microsoft .NET Framework segítségével, a platform által kínált adatbázis-kezelési lehetőségek közül válogat. A cikk összehasonlítja a nyelvben már régóta jelen lévő alacsony szintű SQL parancsokkal végzett megvalósítást a később beépített, de szintén elterjedt objektumrelációs modell keretrendszerrel (azaz az Entity Framework-kel) történő megvalósítással, majd elemzi a kapott eredményeket.

Abstract

The management and storage of data is an essential component in the development of business applications. This is most often implemented by developers with some kind of relational database manager. Receiving and updating data from external sources is a task that often occurs during typical work. The authors of the article are looking for an answer to the question of how to receive this (sometimes very large amount) of data as quickly as possible. The presented case study uses the Microsoft .NET Framework to select from among the database management options offered by the platform. The article compares the implementation with low-level SQL commands, which have been present in the programming language for a long time, with the implementation with the later built-in but also widespread object relational model framework (using the Entity Framework), and then analyzes the obtained results.

* Kapcsolattartó szerző.
E-mail cím: kaczursandor@gmail.com

1. Algoritmusok hatékonyságának áttekintése

Algoritmusok hatékonyságának elemzéséhez, összehasonlításukhoz a cél(ok) egyértelmű meghatározása után követelményrendszer állítható fel. A fejlesztőeszköztől, programozási nyelvtől, implementációtól függő szempontokat nem vizsgáljuk, valamint a grafikus felhasználói felületen történő tevékenységek sem kerülnek említésre. Az alábbi szempontok adódnak: végrehajtási idő, helyfoglalás, bonyolultság [7, 10, 19].

A végrehajtási idő mérhető a végrehajtott utasítások számával, függ a hardvertől. Nem feltétlenül függ a bemenő adatok számától. Lehet minimális, átlagos és maximális.

A helyfoglalás a változók, adatszerkezetek tárigényével mérhető, függ az adat- és programábrázolástól, így nyelv- és implementációfüggő is. Kifejezhető a programkód méretével, illetve a memóriában, háttértáron elfoglalt helyigénnyel is. A helyfoglalás csökkentése általában növeli a végrehajtási időt.

A bonyolultság lehet globális és lokális. Globális, ha az algoritmus egészének megértése szükséges a csökkentéséhez. Lokális, ha kódoptimalizálással (legalább részben) megoldható. Szintén függ az adat- és programábrázolástól, így nyelv- és implementációfüggő. Objektív módon kevésbé definiálható. Csoportosítható logikai és szerkezeti bonyolultságként. Matematikai alapjait [14, 16] tartalmazza.

Az elemzés, mérés során mindig meg kell határozni az indifferens paramétereket, azokat a szempontokat, amelyek nem számítanak, nincsenek hatással a mérésre, ezeket tudatosan kihagy-(hat)juk. Néhány példa: telepítési idő, konfigurálási idő, mintaadatok előkészítése és konvertálása, az eredmények elemzése, az eredmények grafikus megjelenítő képessége, az archiválás képessége szabványos formátumokban, a többféle szoftververzió, az eltérő platformokból illetve programozási nyelvekből adódó jelentős különbségek, a szükséges memory dump eszközök és debuggerek közötti eltérések. Általános megközelítésben szintén nem lényeges, hogy elérhető-e a szoftverekhez, fejlesztői környezetekhez, különböző nyelvi változatokhoz, verziókhoz tartozó dokumentáció, az adatfeldolgozó rész elméleti hátterét tartalmazó publikáció vagy csupán különféle programozási nyelveken (C++, Java, C#, Python) áll rendelkezésre forráskód. Egy hatékonyság szempontjából elemzett szoftver építhet más szoftvercsomagokra, teljes utasításkészlet átvételével, adatdefiníciós rész meghivatkozásával, konvertáló és I/O műveletek importálásával. Ezek a kereszthivatkozások lényegesen csökkentik a strukturális redundanciát, de lényegesen növelik a tervezéshez és megértéshez szükséges absztrakciós szintet, egy-egy algoritmus bonyolultságát, valamint nehezítik az implementált forráskód karbantarthatóságát, korlátozhatják annak továbbfejlesztési lehetőségeit [11, 12, 13].

2. Feladatspecifikáció

Egy .NET Framework 4.8 és SQL Server LocalDB szoftverkörnyezetben működő konzolos alkalmazás megvalósítja tömeges adatok átvételét DBF formátumú, anonimizált személyes adatot tartalmazó, egytáblás relációs adatbázisból, miközben mérésre kerül a végrehajtási idő és a memóriabeli helyfoglalás. Elvárás az objektumorientált tervezés.

3. Tervezés

A tömeges adatok esetében beszúrás (`INSERT INTO`) vagy aktualizálás (`UPDATE`) művelet történik, adott tulajdonságtól vagy állapottól (létezik vagy nem létezik) függően. Az előző mondatban két darab kizáró vagy művelet értendő. A teszt adatbázisban 4780 db releváns rekord található. Az anonomizált személyes adatok HR nyilvántartásból származnak. Hasonló adatok lehetnének még: cafeteria nyilvántartásbeli alkalmazotti adatok, webáruházbeli ügyféladatok, tanulmányi rendszerekben vagy e-learning keretrendszerekben lévő diákok, hallgatók, felhasználók adatai.

A hatékonyság fenti dimenziói közül mérésre kerül a végrehajtási idő és a memóriabeli helyfoglalás. Lényeges követelmény, hogy a mérések elkülönített módon, egymástól függetlenítve működjenek. Másképpen fogalmazva: a mérések ne befolyásolják egymást és a végrehajtás sorrendje se számítson.

3.1. Megvalósítandó stratégiák, tesztesetek

A program alapvetően kétféle szemléletmódot, megközelítést, koncepciót, a továbbiakban stratégiát alkalmaz, és négy tesztesetet tartalmaz. A kétféle stratégia közül az egyik az objektum-relációs modell keretrendszer, a másik az alacsony szintű SQL parancsok használata. Az előbbi stratégia három, az utóbbi egy tesztesetet jelent. A stratégia ebben az esetben a megközelítést, a szemléletmódot, az adatok modellezésének és leképezésének, az adatszerkezetek konstrukciós és szelekciós műveleteinek különböző absztrakciós szintjeit jelöli, amelyek jelentős különbségeket takarnak [1, 6, 17].

A négy tesztesetre a következő azonos paraméterek, körülmények jellemzők:

- mindegyik teszteset azonos feladatot old meg,
- a szoftverkörnyezet független az alkalmazott elvtől, módszertől, megvalósítástól,
- a szoftverkörnyezet: Visual Studio .NET Framework 4.8 és SQL Server LocalDB,
- a fájlkezelés, adatbázis-kezelés és a memóriafogyasztás mérése miatt szükséges a kötelező kivételkezelés erőforrás-foglaló szerkezetet alkalmazása,
- elvárás a logikus és következetes elnevezések és a metódusok egységes paraméterezésének használata.

Az alapvetően kétféle stratégia négy tesztesetből áll [2, 18]:

- EFRowByRow: Entity framework with row-by-row processing.
  Objektumrelációs leképezés. Kliensorientált megközelítés. A rekordokon egyesével halad végig. Az elemi művelet egy rekord feldolgozása. Egyszerűen kódolható. Azt várjuk tőle, hogy hosszú ideig fut és kevés memóriát használ.
- EFAllAtOnce: Entity framework with automatic batch processing, committing all records at once.
  Objektumrelációs leképezés. Automatikus kötegelt foldolgozás. Tranzakció- és kliensorientált megközelítés. A műveletek sikeres lezárása egy lépésben történik. Ez is szintén egyszerűen kódolható. Várhatóan gyorsan fut, de jelentősen több memóriát foglal.
- EFFractionAtOnce: Entity framework with automatic batch processing, committing 100 records at once.
  Objektumrelációs leképezés. Automatikus kötegelt foldolgozás. Tranzakció- és kliensorientált megközelítés. A műveletek sikeres lezárása kisebb egységekben (szegmens, fragment, részhalmaz, például százasával) történik. Ez nehezebben kódolható. Az előző két stratégia eredményei közé várjuk az eredményeit.
- SqlBulkMerge: SQL bulk copy to temp table and merge to final destination.
  Alacsony szintű, beépített SQL utasításokat használ. Ez gyökeresen más megközelítés, ahol a feladat nagy részét delegáljuk az adatbázis-szerverre. Szélsőségesen gyors futást várunk tőle, nagyon alacsony memóriahasználat mellett.

3.2. Objektumorientált tervezés

Az objektumorientált tervezés az alábbiak szerint valósul meg. A program hét osztályból áll, amelyeket a `MassInsertUpdateTest` névtér kapcsol össze. Az osztályok feladatai koncepcionális szinten:

- A `Program` osztály `Main()` metódusa a belépési pont. Modell és vezérlő szerepe van. Létrehozza a `DataTable` típusú `data` objektumot, az adatbázis betöltésével, parszolásával, valamint a `Process` típusú `process` objektumot a memóriafogyasztás/tárigény méréséhez. `IStrategie` típusú generikus `tests` listát épít a négy tesztesetből, egységesen kezelve azokat, amelyeket a `RunTests()` metódus hajt végre. `DeleteDataFromDb()` és `CollectGarbage()` metódusaival biztosítja az adatbázis és a memória alaphelyzeteit, amellyel függetlenné válnak az egymást követő tesztesetek. Tartalmaz két függvényt az eredmények formázásához, beszédes neveik: `GetFormattedTime()` és `GetFormattedMemoryUsage()`. Feladata a konzolos kiírás, a teszteredmények szöveges kommunikációjának megvalósítása.

- A `DbfParser` osztály az adatforrás paramétereit tartalmazza. Felelős a karakterkódolás helyes működéséért, a megfelelő adattípusok használatáért és a közöttük szükséges konverzió megvalósításáért (ezek a műveletek többnyire numerikus és dátum típusokhoz kötődnek). `ReadDBF()` függvénye a paraméterként átvett DBF fájlt betölti `DataTable` típusú memóriabeli adatszerkezetbe. A visszafelé irány megvalósítása nem szükséges.

A `MassInsertUpdateTest.Strategies` névtér összefogja a négy tesztesetet és a közös viselkedésüket biztosító `IStrategie` interfészt, amelyet mindegyik implementál. Az interfész négy metódusfejet (szignatúrát) definiál. A `GetName()` függvény visszaadja a stratégia elnevezését, `GetProcessingTime()` függvény a teszteset végrehajtási idejét, a `GetMemoryUsage()` függvény a felhasznált memóriabeli tárhely méretét adja meg, miközben a `DoJob()` eljárás végrehajtja a tesztesetet. Az interfész implementálása során mindezek négyféleképpen valósulnak meg, az adott stratégiának/tesztesetnek megfelelően.

Végül a `MassInsertUpdateTest.Entities` névtérben található `Jogviszony` osztály az adatforrás (`JOGVISZONY` adatbázisbeli tábla) egyetlen rekordjának memóriabeli objektummá való objektumrelációs leképezését (mapping) valósítja meg – egyirányú módon, az adatbázisból a memóriába irányból közelítve. A visszafelé irány megvalósítása itt sem szükséges.

4. Implementáció

A forráskód lényeges részei következnek a konzolos kiírásoktól többnyire eltekintve.

4.1. Előkészítés

Az adatbázis betöltése a memóriába:

```
DataTable data=DbfParser.ReadDBF(".\\TestDb\\TISZT2.DBF")
```

A négy teszteset hozzáadása a generikus listához:

```
List<IStrategie> tests=new List<IStrategie>() {
  new EFRowByRow(), new EFAllAtOnce(),
  new EFFractionAtOnce(), new SqlBulkMerge() };
```

A tesztadatbázist alaphelyzetbe állító, a táblából adatokat törlő metódus:

```
private static void DeleteDataFromDb() {
  using(TestDatabaseContext context=new TestDatabaseContext()) {
    context.Database.ExecuteSqlCommand(
      "TRUNCATE TABLE JOGVISZONY");
  }
}
```

A memóriát alaphelyzetbe állító, szemétgyűjtőt hívó metódus. A `Collect()` metódus paraméterei biztosítják, hogy az összes, már nem használt memóriaterület felszabadításra és töredezettség-mentesítésre kerül (generációk tekintetében is [4, 5]), valamint a végrehajtás a folyamat befejeződéséig blokkolt.

```
private static void CollectGarbage() {
  GC.Collect(2, GCCollectionMode.Forced, true, true);
}
```

A vezérlést megvalósító, a teszteseteket futtató metódus:

```
private static void RunTests(List<IStrategie> tests) {
  int i=1;
  foreach(var test in tests) {
    DeleteDataFromDb();
    CollectGarbage();
    Console.WriteLine(
      string.Format("[+] Starting test method {0}: {1}", i++,
      test.GetName()));
    test.DoJob(data, process);
```

```
        Console.WriteLine(
          string.Format("[.] Execution time was: {0}",
          GetFormattedTime(test.GetProcessingTime())));
        Console.WriteLine(
          string.Format("[.] Method used {0} bytes of memory.",
          GetFormattedMemoryUsage(test.GetMemoryUsage())));
      }
    }
```

4.2. Az `EFRowByRow` stratégia alkalmazása

```
public void DoJob(DataTable data, Process process) {
  process.Refresh();
  memoryUsedBefore=process.PrivateMemorySize64;
  using(TestDatabaseContext context=new TestDatabaseContext()) {
    DateTime startTime=DateTime.Now;
    foreach(DataRow item in data.Rows) {
      if(System.DBNull.Value!=item["BELEPDAT"]) {
        Jogviszony jv=new Jogviszony(item);
        string error=jv.IsDataValid();
        if(string.IsNullOrEmpty(error)) {
          Jogviszony oldJv=context.Jogviszony.
            FirstOrDefault(a=>a.SzemAdoazon==jv.SzemAdoazon);
          if(null!=oldJv) {
            oldJv.HrAzon=jv.HrAzon;
            oldJv.CsNev=jv.CsNev;
            oldJv.UNev=jv.UNev;
            oldJv.JogvKezdete=jv.JogvKezdete;
            oldJv.JogvVege=jv.JogvVege;
            context.SaveChanges();
          }
          else {
            context.Jogviszony.Add(jv);
            context.SaveChanges();
          }
        }
      }
    }
    DateTime endTime=DateTime.Now;
    processingTime=(endTime-startTime).TotalMilliseconds;
    process.Refresh();
    memoryUsedAfter=process.PrivateMemorySize64;
  }
}
```

A továbbiakban az egyes stratégiák esetén csupán a lényeges különbségeket emeljük ki.

4.3. Az `EFAllAtOnce` stratégia alkalmazása

```
  List<string> existingTaxNumbers=context.Jogviszony.
    AsNoTracking().Select(i=>i.SzemAdoazon).ToList();
  foreach(DataRow item in data.Rows) {
    Jogviszony jv=new Jogviszony(item);
    string error=jv.IsDataValid();
    if(string.IsNullOrEmpty(error)) {
      if(existingTaxNumbers.Contains(jv.SzemAdoazon)) {
        Jogviszony oldJv=context.Jogviszony.
          FirstOrDefault(a=>a.SzemAdoazon==jv.SzemAdoazon);
```

4.4. Az `EFFractionAtOnce` stratégia alkalmazása

```
  const int FRACTION=100;
//
  int recCount=0;
//
  TestDatabaseContext context=new TestDatabaseContext();
  List<string> existingTaxNumbers=context.Jogviszony.
    AsNoTracking().Select(i=>i.SzemAdoazon).ToList();
  foreach(DataRow item in data.Rows) {
    Jogviszony jv=new Jogviszony(item);
    string error=jv.IsDataValid();
    if(string.IsNullOrEmpty(error)) {
      if(existingTaxNumbers.Contains(jv.SzemAdoazon)) {
//
      }
      recCount++;
    }
    if(recCount==FRACTION) {
      context.SaveChanges();
      recCount=0;
      context.Dispose();
      context=new TestDatabaseContext();
    }
//
  }
  context.SaveChanges();
  context.Dispose();
  context=null;
//
```

4.5. Az `SqlBulkMerge` stratégia alkalmazása

```
  using(TestDatabaseContext context=new TestDatabaseContext()) {
    DbConnection conn=context.Database.Connection;
//
    if(conn is SqlConnection sqlConn &&
        conn.State==ConnectionState.Open) {
      SqlBulkCopy bulkCopy=new SqlBulkCopy(sqlConn);
      bulkCopy.DestinationTableName="#NewJogviszony";
      bulkCopy.ColumnMappings.Add("SORSZAM", "HrAzon");
      bulkCopy.ColumnMappings.Add("ADOSZAM", "SzemAdoazon");
      bulkCopy.ColumnMappings.Add("TAJ", "Tajszam");
//
      data.Columns.Add(new DataColumn("CSNEV", typeof(string)));
      data.Columns.Add(new DataColumn("UNEV", typeof(string)));
//
      DbCommand cmd=conn.CreateCommand();
      cmd.CommandText="CREATE TABLE #NewJogviszony (//
//
      cmd.ExecuteNonQuery();
      foreach(DataRow item in data.Rows) {
//
      }
      bulkCopy.WriteToServer(data);
```

```
        cmd.CommandText=
          "MERGE Jogviszony AS Trg USING #NewJogviszony AS Src ON
           (Trg.SzemAdoazon=Src.SzemAdoazon) WHEN MATCHED THEN
           UPDATE SET Trg.Tajszam=Src.Tajszam, Trg.CsNev=Src.CsNev,
//
             Trg.HrAzon=Src.HrAzon WHEN NOT MATCHED BY TARGET THEN
           INSERT (SzemAdoAzon, Tajszam, CsNev…) VALUES
             (Src.SzemAdoazon, Src.Tajszam, Src.CsNev…);";
        processed=cmd.ExecuteNonQuery();
        cmd.CommandText="DROP TABLE #NewJogviszony";
        cmd.ExecuteNonQuery();
        cmd.Dispose();
        conn.Close();
```

5. Teszteredmények

A tervezés során megfogalmazott előzetes elvárásokat igazolták a program végrehajtása során kapott teszteredmények (1. táblázat). Attól függően, hogy adott problémától, környezettől, paraméterektől függően mire szükséges optimalizálni, az adatok segíthetnek kiválasztani a tömeges adatimport során használandó megfelelő stratégiát.

1. táblázat. Teszteredmények

Stratégia	*Végrehajtási idő*	*Tárigény*
EFRowByRow	31,8 mp	824 kB
EFAllAtOnce	4,7 mp	28,25 MB
EFFractionAtOnce	17,7 mp	1,01 MB
SqlBulkMerge	0,2 mp	60 kb

6. Összegzés

A tesztelt négyféle stratégia eredményei alapján röviden megválaszolva a cikk címében feltett kérdést:

- EF objektumrelációs modellezéssel/leképezéssel az EFFractionAtOnce stratégiát érdemes használni. Általában nem kifejezetten egyféle hatékonysági szempontot kell figyelembe venni. Ez megfelelő kompromisszumot jelenthet a közepes végrehajtási idejével és az alacsony tárigényével.
- Objektumrelációs leképezés helyett alacsony szintű SQL utasításokat érdemes alkalmazni. Az SqlBulkMerge stratégia „mindent visz”.

Az EF megközelítés könnyen kódolható, de csak akkor hasznos igazán, ha csupán néhány rekord manipulálására használjuk. Tömeges adatokkal végzett tevékenységekre az alacsony szintű SQL utasítások alkalmasak inkább [7, 17].

Ismerve a felsőoktatás mérnökinformatikus képzésének kapcsolódó tantárgyi tematikáit és tantárgyi egymásra épülését, markáns különbségek fogalmazhatók meg. Az EF, azaz az objektumrelációs leképezés jóval absztraktabb témakör. Ezért jóval tovább tart eljuttatni a tananyag elméleti részén keresztül annak gyakorlati alkalmazásáig a hallgatókat. Tehát magasabb a belépési küszöb, de az EF használata egyszerűbb. Az alacsony szintű SQL megközelítés kevésbé absztrakt. Hamarabb van sikerélménye a hallgatóknak, mert egyszerűbb mintapéldák alapján modulárisan felépíthető belőle (alulról felfelé haladva) egy alkalmazás, ha csupán karbantartásra és lekérdezésre használjuk, nem adatmodellezésre.

Általánosságban kijelenthető [7, 9, 14, 18]:

- Érdemes programozási nyelvtől független ötletekből kiindulva tervezni és a későbbi implementáció során bevetni a nyelv lehetőségeit, újdonságait, összetett eszköztárát, képességeit.
- Elosztott alkalmazás esetén a funkciók szétosztásánál a lehetőségek, előre definiált paraméterek, határok mentén érdemes egyensúlyozni a szerver és a kliens között.

Köszönetnyilvánítás

A szerzők köszönetet mondanak Kállai Miklósnak, aki egy gyakorlati probléma bemutatásával inspirálta ennek a projektnek, esettanulmánynak létrejöttét, továbbá Endrődi Tamásnak az alacsony szintű SQL Server megvalósításhoz nyújtott támogatásáért.

Az it-tanfolyam.hu oktatói csapata köszönetet mond a Java adatbázis-kezelő tanfolyam [8] alumni hallgatóinak, akik hosszú évek óta folyamatosan hasznos észrevételeket tesznek a felhasznált projektjeink, esettanulmányaink, oktatóprogramjaink kapcsán és követik szakmai blogunkat a tanfolyam tematikájához kötődő kategóriában [9] is.

Irodalomjegyzék

[1] Benedek, Z., Levendovszky, T., Kovács, F: Szoftvertechnikák előadás, Adatbázis-kezelés alapjai, ADO.NET, Automatizálási és Alkalmazott Informatikai Tanszék, BME, http://kompabt.web.elte.hu/enripted.pdf, 2023.05.28.
[2] Erdélyi, T., Smulovics, P.: Objektum-relációs leképezés a .NET framework segítségével, szakdolgozat, Automatizálási és Alkalmazott Informatikai Tanszék, BME, https://nws.niif.hu/ncd2002/docs/ehu/62/index.html, 2023.05.28.
[3] T. FitzMacken: Introduction to Working with a Database in ASP.NET Web Pages (Razor) Sites, https://learn.microsoft.com/en-us/aspnet/web-pages/overview/data/5-working-with-data, 2023.05.28.
[4] Fundamentals of garbage collection, In: Microsoft Technical documentation, https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals, 2023.05.28.
[5] GC.Collect Method, In: Microsoft Technical documentation, https://learn.microsoft.com/en-us/dotnet/api/system.gc.collect?view=netframework-4.8, 2023.05.28.
[6] A. Gruca, P. Podsiadło: Performance Analysis of .NET Based Object–Relational Mapping Frameworks, 10th International Conference, BDAS Ustron, Poland, Springer International Publishing, DOI: 10.1007/978-3-319-06932-6, pp.40-49, 2014
[7] Iványi, A.: Informatikai algoritmusok I., II. ELTE, 2004., 2005., ISBN 9634637752
[8] Java adatbázis-kezelő tanfolyam, it-tanfolyam.hu landing page, https://it-tanfolyam.hu/java-adatbazis-kezelo-tanfolyam/, 2023.08.21.
[9] Java adatbázis-kezelő tanfolyam, it-tanfolyam.hu szakmai blog bejegyzések kategóriája, https://it-tanfolyam.hu/kategoria/javaab/, 2023.08.21.
[10] Kaczur S.: A hozzárendelési feladat két megoldási módszerének összehasonlító elemzése, Informatika, A GDF Közleményei, Budapest, XI. évfolyam 2. szám, 2009, ISSN 1419-2527, p. 24-27
[11] Kaczur, S.: Néhány nyílt forráskódú EKG analizáló algoritmus hatékonysága, A Tudomány Hete a Dunaújvárosi Főiskolán, Dunaújváros, Dunaújvárosi Főiskola, 2011.11.7-11.
[12] S. Kaczur: The efficiency of data visualization software for Physiobank, Informatika, A GDF Közleményei, Budapest, XV. évfolyam 1. szám, 2013, ISSN 1419-2527, p. 17-19
[13] Kaczur, S.: A PhysioBank fiziológiai jelfeldolgozó szoftvereinek hatékonysága, Informatika a felsőoktatásban 2014 Konferencia kiadvány, Debrecen, Debreceni Egyetem Informatikai Kar, 2014, ISBN 978-963-473-712-4, p. 54-59
[14] Mágoriné, H. Á.: Algoritmusok és adatszerkezetek, JGYF Kiadó, Szeged, 2000, ISBN 963 9167 36 3
[15] G. Procaccianti, P. Lago, W. Diesveld: Energy Efficiency of ORM Approaches: An Empirical Evaluation, Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement ESEM, Ciudad Real, Spain, ACM, DOI: 10.1145/2961111.2962586, pp.185–198, 2016
[16] Szlávi, P., Zsakó, L., Temesvári, T.: Módszeres programozás: A programkészítés technológiája, Mikrológia 21., ELTE, 2000
[17] P. Van Zyl, D.G. Kourie, A. Boake: Comparing the performance of object databases and ORM tools, Proceedings of the 2006 Annual research conference of the SAICSIT on IT research in developing countries, SAICSIT 2006, Somerset West, South Africa, DOI: 10.1145/1216262.1216263, pp. 1–11, 2006
[18] W. Wiphusitphunpol, T. Lertrusdachakul: Fetch performance comparison of object relational mapper in. NET platform. In 2017 IEEE 14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology ECTI-CON, DOI: 10.1109/ECTICon.2017.8096264, pp. 423-426, 2017
[19] Zsakó, L.: Módszeres programozás: Hatékonyság, Mikrológia 6., ELTE, 1999