Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Szoftvertechnikák laborgyakorlat

5. mérés

Adatkezelés

ADO.NET kapcsolat alapú modell

Ez az oktatási segédanyag a Budapesti Műszaki és Gazdaságtudományi Egyetem oktatója által kidolgozott szerzői mű. Kifejezett felhasználási engedély nélküli felhasználása szerzői jogi jogsértésnek minősül.

A gyakorlatot kidolgozta: Kaszó Márk, Simon Gábor (simon.gabor@aut.bme.hu)
Utolsó módosítás ideje: 2020.04.10.

Tartalom

A gyakorlat célja

A kapcsolódó előadások:

- Adatkezelés, ADO.NET alapismeretek

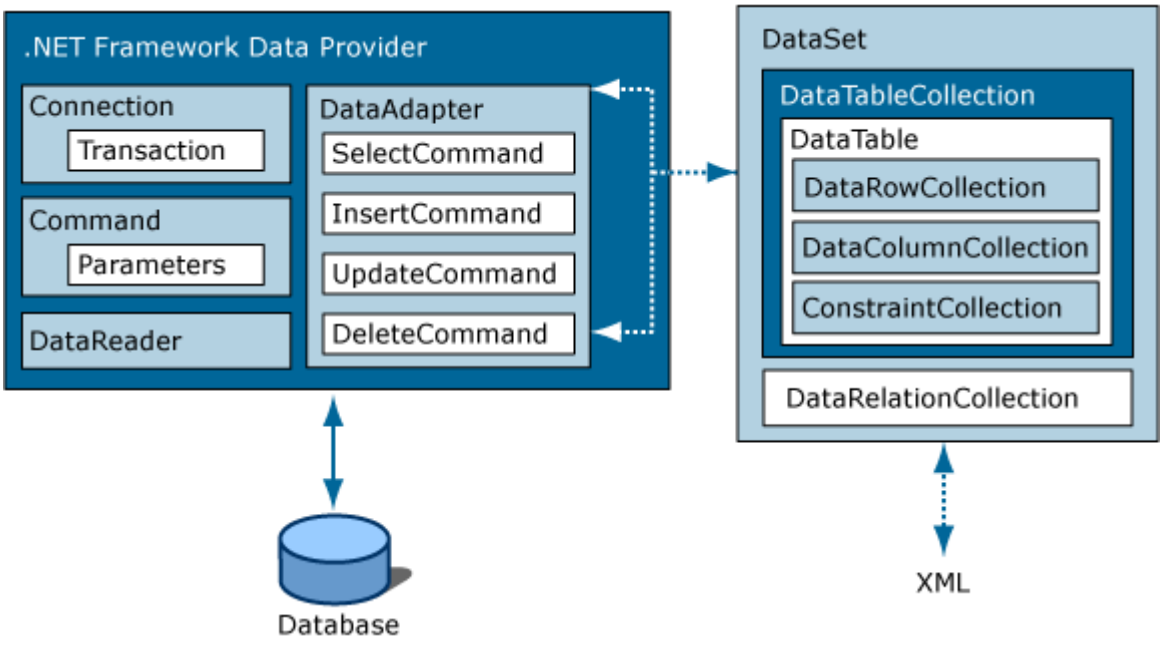
A gyakorlat célja az ADO.NET programozási modelljének megismerése és a leggyakoribb adatkezelési problémák, buktatók szemléltetése alapvető CRUD műveletek megírásán keresztül.

Bevezetés

Megjegyzés laborvezetőknek: ezt a fejezetet laboron nem kell a leírtaknak megfelelő részletességgel ismertetni, a fontosabb fogalmakat azonban mindenképpen ismertessük röviden.

ADO.NET

Alacsony szintű adatbáziskezelésre a .NET Frameworkben az ADO.NET áll rendelkezésre, segítségével relációs adatbázisokat tudunk elérni.



1. ábra - ADO.NET architektúra

Az alapfolyamat a következő:

1. Kapcsolat létrehozása az alkalmazás illetve az adatbázis kezelő rendszer között (`Connection` objektum)
2. A futtatandó SQL utasítás összeállítás (`Command` objektum felhasználásával)
3. Utasítás futtatása
4. Lekérdezések esetén a visszakapott rekordhalmaz feldolgozása (`DataReader`, opcionálisan `DataSet`)
5. Kapcsolat lezárása

Megjegyzés: az ábrán látható DataAdapter és DataSet alkalmazása csak a kapcsolat nélküli modell esetén használatos.

Kapcsolatalapú modell: Lényege az, hogy az adatbáziskapcsolatot végig nyitva tartjuk, amíg az adatokat lekérdezzük, módosítjuk, majd a változtatásokat az adatbázisba visszaírjuk. A megoldásra *DataReader* objektumokat használhatunk (lásd később). A megoldás előnye az egyszerűségében rejlik (egyszerűbb programozási modell és konkurenciakezelés). A megoldás hátránya, hogy a folyamatosan fenntartott hálózati kapcsolat miatt skálázhatósági problémák adódhatnak. Ez azt jelenti, hogy az adatkezelőhöz történő nagyszámú párhuzamos felhasználói hozzáférés esetén folyamatosan nagyszámú adatbázis kapcsolat él, ami adatkezelő rendszerek esetén a teljesítmény szempontjából költséges erőforrásnak számít. Így a fejlesztés során célszerű arra törekedni, hogy az adatbázis kapcsolatokat mielőbb zárjuk le.

- A modell előnyei:
    - Egyszerűbb a konkurencia kezelése
    - Az adatok mindenhol a legfrissebbek

  Megjegyzés: ezek az előnyök akkor jelentkeznek, ha az adatbázis hozzáférés az adatkezelő szigorú zárakat használ – ezt mi a hozzáférés során megfelelő tranzakció izolációs szint megadásával tudjuk szabályozni. Ennek technikái későbbi tanulmányok során kerülnek ismertetésre).
- Hátrányok:
    - Folyamatos hálózati kapcsolat
    - Skálázhatóság

Kapcsolat nélküli modell: A kapcsolatalapú modellel ellentétben az adatok megjelenítése és memóriában történő módosítása során nem tartunk fent adatbázis kapcsolatot. Ennek megfelelően a főbb lépések a következők: a kapcsolat felvételét és az adatok lekérdezését követően azonnal bontjuk a kapcsolatot. Az adatokat ezt követően tipikusan megjelenítjük és lehetőséget biztosítunk a felhasználónak az adatok módosítására (rekordok felvétele, módosítása, törlése igény szerint). A módosítások mentése során újra felvesszük az adatkapcsolatot, mentjük az adatbázisba a változtatásokat és zárjuk a kapcsolatot. Természetesen a modell megköveteli, hogy a lekérdezése és a módosítások visszaírása között – amikor nincs kapcsolatunk az adatbázissal – az adatokat és a változtatásokat a memóriában nyilvántartsuk. Erre az ADO.NET környezetben nagyon kényelmes megoldást nyújt a `DataSet` objektumok alkalmazása.

A kapcsolatnélküli modell használata a következőkkel jár:

- Előnyök
    - Nem szükséges folyamatos hálózati kapcsolat
    - Skálázhatóság
- Hátrányok
    - Az adatok nem mindig a legfrissebbek
    - Ütközések lehetségesek

Megjegyzés: Számos lehetőségünk van arra, hogy az objektumokat és kapcsolódó változásokat nyilvántartsuk a memóriában. A DataSet csak az egyik lehetséges technika. De használhatunk erre a célra közönséges objektumokat, illetve ezek menedzselését megkönnyítő ADO.NET-nél korszerűbb .NET technológiákat (pl. Entity Framework).

A kapcsolat alapú modell

A mérés keretében a kapcsolat alapú modellt ismerjük meg. Az adatbázissal való kommunikációnak ebben a modellben három fő összetevője van:

- `Connection`
- `Command`
- `Data Reader`

Ezek az összetevők egy-egy osztályként jelennek meg, adatbáziskezelőfüggetlen részük a System.Data.dll-ben található `DbConnection`, `DbCommand`, illetve `DbDataReader` néven. Ezek absztrakt osztályok, az adatbáziskezelők gyártóinak feladata, hogy ezekből leszármazva megírják a konkrét adatbáziskezelőket támogató változatokat. A Microsoftt SQL Servert támogató változatok szintén a System.Data.dll-ben, „Sql" prefixű osztályokban találhatók (pl. `SqlConnection`).

A többi gyártó külön dll-(ek)be teszi a saját változatát, az így létrejött komponenst data providernek nevezik. Található még System.Data.dll-ben Oracle adatbáziskezelőt támogató provider is, azonban ez már elavult. Néhány további gyártó ADO.NET adatbázis providere a teljesség igénye nélkül: ODP.NET (Oracle), MySqlConnector/Net (MySQL), Npgsql (PostgreSQL), System.Data.SQLite (SQLite)

Mindhárom ADO.NET összetevő támogatja a Dispose mintát, így `using` blokkban használható – használjuk is így, amikor csak tudjuk. Az adatbáziskezelő általában másik gépen található, mint ahol a kódunk fut (a mérés során pont nem ☺), így tekintsünk ezekre, mint távoli hálózati erőforrásokra.

Connection

Ez teremti meg a kapcsolatot a programunk, illetve az adatbáziskezelő-rendszer között

Inicializálásához szükség van egy *connection string*-re, ami a kapcsolat felépítéséhez szükséges adatokat adja meg a driver számára. Adatbázisgyártónként eltérő belső formátuma van (bővebben: http://www.connectionstrings.com).

Új `Connection` példányosításakor nem biztos, hogy tényleg új kapcsolat fog létrejönni az adatbázis felé, a driverek általában connection pooling-ot alkalmaznak, hasonlóan, mint a thread pool esetében, újrahasználhatják a korábbi (éppen nem használt) kapcsolatokat.

Command

Ennek segítségével vagyunk képesek „utasításokat" megfogalmazni az adatbázis kezelő számára. Ezeket SQL nyelven kell megfogalmaznunk.

A Command-nak be kell állítani egy kapcsolatot – ezen keresztül fog a parancs végrehajtódni. A parancsnak különböző eredménye lehet, ennek megfelelően különböző függvényekkel sütjük el a parancsot:

- `ExecuteReader`: Eredményhalmaz (result set) lekérdezése
- `ExecuteScalar`: Skalár érték lekérdezése
- `ExecuteNonQuery`: Nincs visszatérési érték (Pl: INSERT), viszont a művelet következtében érintett rekordok számát visszakapjuk

Data Reader

Ha a parancs eredménye eredményhalmaz, akkor ennek a komponensnek a segítségével tudjuk az adatokat kiolvasni. Az eredményhalmaz egy táblázatnak tekinthető, a Data Reader ezen tud kurzoros módszerrel soronként végignavigálni (csak előrefelé!). A kurzor egyszerre egy soron áll, ha a sorból a szükséges adatokat kiolvastuk a kurzort egy sorral előre léptethetjük. Csak az aktuális sorból tudunk olvasni. Kezdetben a kurzor nem az első soron áll, azt egyszer léptetnünk kell, hogy az első sorra álljon.

Megjegyzés: navigálás kliens oldalon történik a memóriában, nincs köze az egyes adatkezelők által támogatott kiszolgáló oldali kurzorokhoz.

Telepítendő komponensek

A labor során az SQL Server Object Explorer-t fogjuk használni az adatbázis objektumok közötti navigálására és a lekérdezések futtatására. Ehhez szükség lehet az **SQL Server Data Tools** komponensre, amit legegyszerűbben az „Individual Components” oldalon tudunk telepíteni a Visual Studio Installer-ben.

Feladat 1 – Adatbázis előkészítése

Elsőként szükségünk van egy adatbáziskezelőre. Ezt valós környezetben dedikált szerveren futó, adatbázis adminisztrátorok által felügyelt, teljesértékű adatbáziskezelők jelentik. Fejlesztési időben, lokális teszteléshez azonban kényelmesebb fejlesztői adatbáziskezelő használata.

A Visual Studio telepítésének részeként kapunk is egy ilyen adatbázis engine-t, ez a LocalDB, ami a teljesértékű SQL Server egyszerűsített változata. Főbb tulajdonságai:

- nem csak a Visual Studio-val, hanem külön is telepíthető
- az adatbázismotor szinte teljes mértékben kompatibilis a teljesértékű Microsoft SQL Serverrel
- alapvetően arról a gépről érhető el, amire telepítettük
- több példány is létrehozható igény szerint, a példányok alapvetően a létrehozó operációs rendszer felhasználója számára érhető el (igény esetén megosztható egy példány a felhasználók között)
- a saját példányok kezelése (létrehozás, törlés, stb.) nem igényel adminisztrátori jogokat

A példányok kezelésére az sqllocaldb parancssori eszköz használható. Néhány parancs, amit az *sqllocaldb* után beírva alkalmazhatunk (ezeket laboron nem kell elmondani), csak a lehetőséget említsük meg:

Parancs	Leírás
info	az aktuális felhasznááló számára látható példányok listája
create „locdb”	új példány létrehozása „locdb” névvel
delete „locdb”	„locdb” nevű példány törlése
start „locdb”	„locdb” nevű példány indítása
stop „locdb”	„locdb” nevű példány leállítása

A Visual Studio is vesz fel, illetve indít LocalDB példányokat, ezért érdemes megnézni, hogy a Visual Studio alapból milyen példányokat lát.

1. Indítsuk el a Visual Studio-t, a View menüből válasszuk az SQL Server Object Explorer-t (SSOE).
2. Nyissuk ki az SQL Server csomópontot, ha alatta látunk további csomópontokat, akkor nyert ügyünk van, nyissuk ki valamelyiket (ilyenkor indul el a példány, ha nincs elindítva, így lehet, hogy várni kell kicsit)
3. Ha nem jelent meg semmi, akkor parancssorból az „`mssqllocaldb info`” parancs megadja a létező példányokat. Válasszuk az SQL Server csomóponton jobbklikkelve az `Add SQL Server` opciót, majd adjuk meg valamelyik létező példányt, pl.: `(localdb)\MSSQLLocalDB`
4. A megjelenő `Databases` csomóponton válasszuk a `New Database` opciót, itt adjunk meg egy adatbázisnevet. (Laboron, mivel több hallgató is használhatja ugyanazt az operációs rendszer felhasználót, így javasolt a neptunkód mint név használata.)
5. Az új adatbázis csomópontján jobbklikkelve válasszuk a `New Query` opciót, ami egy új query ablakot nyit
6. A tárgyhonlapról nyissuk meg vagy töltsük le a `Northwind` adatbázis inicializáló szkriptet (a publikus anyagok között található, „Northwind példaadatbázis” néven)
7. Másoljuk be a teljes szkriptet a query ablakba
8. A szkript elején a megadott helyen írjuk be az adatbázisunk nevét
9. Futassuk le a szkriptet a kis zöld nyíllal (`Execute`)
10. Ellenőrizzük, hogy az adatbázisunk ban megjelentek-e táblák, nézetek
11. Fedezzük fel az SSOE legfontosabb funkcióit (táblák adatainak, sémájának lekérdezése, stb.)

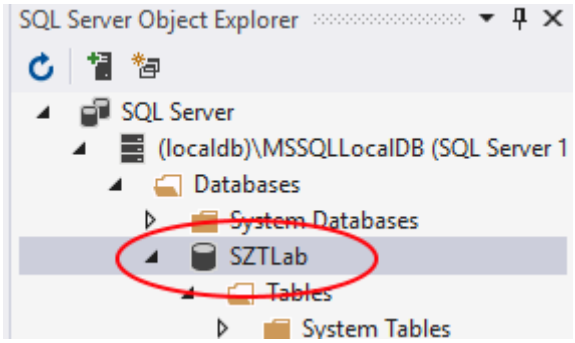
A Visual Studio-ban két eszközzel is kezelhetünk adatbázisokat: a Server Explorer-rel és az SQL Server Object Explorer-rel is. Előbbi egy általánosabb eszköz, mely nem csak adatbázis, hanem egyéb szerver erőforrások (pl. Azure szerverek) kezelésére is alkalmas, míg a másik kifejezetten csak adatbáziskezelésre van kihegyezve. Mindkettő elérhető a View menüből és mindkettő hasonló funkciókat ad adatbáziskezeléshez, ezért ebben a mérésben csak az egyiket (SQL Server Object Explorer) használjuk.

Amikor nem áll rendelkezésünkre a Visual Studio fejlesztőkörnyezet, akkor az adatbázisunk menedzselésére az (ingyenes) SQL Server Management Studio-t tudjuk használni.

Feladat 2 – Lekérdezés ADO.NET SqlDataReader-rel

A feladat egy olyan C# nyelvű konzol alkalmazás elkészítése, ami használja a `Northwind` adatbázis `Shippers` táblájának rekordjait.

1. Hozzunk létre egy C# nyelvű konzolos alkalmazást. A projekt típusa „**Console App (.NET Framework)**” legyen, és **NE** „Console App (.Net Core)”. A projekt neve legyen ADOExample.
2. Keressük ki a connection stringet az SSOE-ből: jobbklikk az adatbáziskapcsolatunkon (pirossal jelölve az alábbi ábrán) -> `Properties`.

3. 

Másoljuk a `Properties` ablakból *Connection String* tulajdonság értékét egy konstans változóba (pl. a Program osztályba).

```
private const string CONN_STRING = @"Data Source=(localdb)\ProjectsV12;Initial
Catalog=xgef0q; Integrated Security=True;…";
```

Megjegyzés: a Data Source kulcs alatt az SQL szerver példány neve, az Initial Catalog kulcs alatt az adatbázis neve szerepel.

Megjegyzés: a @ egy speciális karakter (verbatim indentifier), amit itt arra használunk, hogy a connection string-ben megjelenő backslash karakter (\) ne feloldójelként (escape character) kerüljön értelmezésre.

4. Írjunk lekérdező függvényt, ami lekérdezi az összes szállítót:

```
private static void GetShippers()
{
    using (SqlConnection conn = new SqlConnection(CONN_STRING))
    using (SqlCommand command =
        new SqlCommand("SELECT ShipperID, CompanyName, Phone FROM Shippers", conn))
    {
        conn.Open();
        Console.WriteLine("{0,-10}{1,-20}{2,-20}"
                          ,"ShipperID", "CompanyName", "Phone");
        Console.WriteLine(new string('-',60));
        using (SqlDataReader reader = command.ExecuteReader())
        {
            while (reader.Read())
                Console.WriteLine($@"{
                                        reader["ShipperID"],-10
                                    }{
                                        reader["CompanyName"],-20
                                    }{
                                        reader["Phone"],-20
                                    }");
        }
    }
}
```

A kapcsolat alapú modell folyamata

1. *Kapcsolat, parancs inicializálása*
2. *Kapcsolat megnyitása*
3. *Parancs futtatása*

4. *Eredmény feldolgozása*
5. *Kapcsolat bontása, takarítás*

Néhány megjegyzés a kódhoz:

- *a Data Reader-t a parancs futtatásának eredményeként kapjuk meg, nem pedig közvetlenül példányosítjuk*
- *a parancs futtatása előtt meg kell nyitnunk a kapcsolatot*
- *a Connection példányosításakor nem nyitódik meg a kapcsolat (nem történik hálózati kommunikáció)*
- *a Data Reader Read függvénye mutatja, hogy van-e még adat az eredményhalmazban*
- *a Data Reader-t az eredményhalmazban található oszlopok nevével indexelhetjük – az eredmény object lesz, így, ha konkrétabb típustra van szükségünk cast-olni kell*
- *a fordító nem értelmezi az SQL parancs szövegét (az csak egy string), hanem csak majd az adatbázis, így hibás SQL esetén csak futási idejű kivételt kapunk*
- *figyeljük meg, hogy az adatbázis séma változása esetén, pl. egy oszlop átnevezése, hány helyen kell kézzel átírni string-eket a kódban*
- *$-val prefixelve egy stringkonstanst string interpolációt alkalmazhatunk, azaz közvetlenül a stringkonstansba ágyazhatunk kiértékelendő kifejezéseket (C# 6-os képesség). A $@ segítségével többsoros string interpolációs kifejezéseket írhatunk (a sortörést a {}-k között kell betennünk, különben a kimeneten is megjelenik). Érdekesség: C# 8-tól fölfele bármilyen sorrendben írhatjuk a $ és @ karaktereket, tehát a $@ és a @$ is helyesnek számít.*

5. Hívjuk meg új függvényünket a `Main` függvényből (a `Main` utolsó sorába írjunk be egy Console.ReadKey() utasítást, hogy ne záródjon be azonnal az alkalmazásunk kilépéskor).
6. Próbáljuk ki az alkalmazást. Rontsuk el az SQL-t, és úgy is próbáljuk ki.

Feladat 3– Beszúrás SQL utasítással

1. Írjunk függvényt, ami új szállítót szúr be az adatbázisba:

```
private static void InsertShipper(string companyName, string phone)
{
    using (SqlConnection conn = new SqlConnection(CONN_STRING))
    using (SqlCommand command = new SqlCommand(
        "INSERT INTO Shippers(CompanyName, Phone) VALUES(@name,@phone)",
        conn))
    {
        command.Parameters.AddWithValue("@name",companyName);
        command.Parameters.AddWithValue("@phone", phone);
        conn.Open();
        int affectedRows = command.ExecuteNonQuery();
        Console.WriteLine($"{affectedRows} rows affected");
    }
}
```

Itt olyan SQL-t kell írnunk, aminek az összeállításánál kívülről kapott változók értékeit is felhasználtuk. A string összerakásához egyszerűen a string összefűzés operátort vagy `string.Format`-ot is használhattunk volna, de ez biztonsági kockázatot (SQL Injection – bővebben lásd lentebb) rejt – SOHA!!! ne rakjuk össze az SQL-t stringművelettel. Helyette írjuk meg úgy az SQL-t, hogy ahová a változók értékeit írnánk, oda paraméterhivatkozásokat teszünk. SQL Server esetében a hivatkozás szintaxisa: `@paramnév`.

A parancs elsütéséhez a paraméterek értékeit is át kell adnunk az adatbázisnak, ugyanis az fogja elvégezni a paraméterek helyére az értékek behelyettesítését.

A beszúrási parancs kimenete nem eredményhalmaz - így másik függvénnyel (*ExecuteNonQuery*) kell elsütnünk-, viszont visszaadja beszúrt sorok számát.

2. Hívjuk meg új függvényünket a `Main` függvényből.

```
GetShippers();
InsertShipper("Super Shipper","49-98562");
GetShippers();
```

3. Próbáljuk ki az alkalmazást, ellenőrizzük a konzolban és az SSOE-ben is, hogy bekerült az új sor. SSOE-ben való gyors és kényelmes ellenőrzéshez a `Shippers` tábla context menüjéből válasszuk a „View Data” lehetőséget.

Feladat 4 - Módosítás tárolt eljárással

1. Tanulmányozzuk SSOE-ben a `Product_Update` tárolt eljárás kódját. Ehhez nyissuk le a Programmability alatt található Stored Procedures csomópontot, majd a `Product_Update` tárolt eljárás context menüjéből válasszuk a „View Code” lehetőséget.

   Megjegyzés: A komolyabb adatkezelő rendszerek lehetőséget biztosítanak arra, hogy programkódot definiáljunk magában az adatkezelőben. Ezeket tárol eljárásoknak (stored procedure) nevezzük. A nyelve adatkezelő függő.

2. Írjunk függvényt, ami ezt a tárolt eljárást hívja

```
private static void UpdateProduct(int productID, string productName, int price)
{
    using (SqlConnection conn = new SqlConnection(CONN_STRING))
    using (SqlCommand command = new SqlCommand("Product_Update", conn))
    {
        command.CommandType = CommandType.StoredProcedure;
        command.Parameters.AddWithValue("@ProductID", productID);
        command.Parameters.AddWithValue("@ProductName", productName);
        command.Parameters.AddWithValue("@UnitPrice", price);
        conn.Open();
        int affectedRows = command.ExecuteNonQuery();
        Console.WriteLine($"{affectedRows} rows affected");
    }
}
```

A `Command`-ban a tárolt eljárás nevét kellett megadni és a parancs típusát kellett átállítani, egyébként szerkezetileg hasonlít a beszúró kódra.

3. Hívjuk meg új függvényünket a `Main` függvényből, például az alábbi paraméterezéssel:

```
UpdateProduct(1,"MyProduct",50);
```

4. Próbáljuk ki az alkalmazást, ellenőrizzük a konzolban és az SSOE-ben is, hogy módosult-e az 1-es azonosítójú termék.

Feladat 5 - SQL Injection

1. Írjuk meg a beszúró függvényt úgy, hogy string interpolációval rakja össze az SQL-t

```
private static void InsertShipper2(string companyName, string phone)
{
    using (SqlConnection conn = new SqlConnection(CONN_STRING))
    using (SqlCommand command = new SqlCommand(
        $"INSERT INTO Shippers(CompanyName, Phone)
VALUES('{companyName}','{phone}')",
        conn))
    {
        conn.Open();
        int affectedRows = command.ExecuteNonQuery();
        Console.WriteLine($"{affectedRows} row(s) inserted");
    }
}
```

2. Hívjuk meg új függvényünket a Main függvényből „speciálisan” paraméterezve

```
InsertShipper2("Super Shipper", "49-98562'); DELETE FROM Shippers;--");
```

Úgy állítottuk össze a második paramétert, hogy az lezárja az eredeti utasítást, ezután tetszőleges (!!!) SQL-t írhatunk, végül kikommentezzük az erdeti utasítás maradékát (--).

3. Próbáljuk ki az alkalmazást, hibát kell kapjunk, hogy valamelyik szállító nem törölhető idegen kulcs hivatkozás miatt.

Tehát a `DELETE FROM` is lefutott! Nézzük meg debuggerrel (pl. a conn.Open utasításon állva), hogy mi a végleges SQL (command.CommandText).

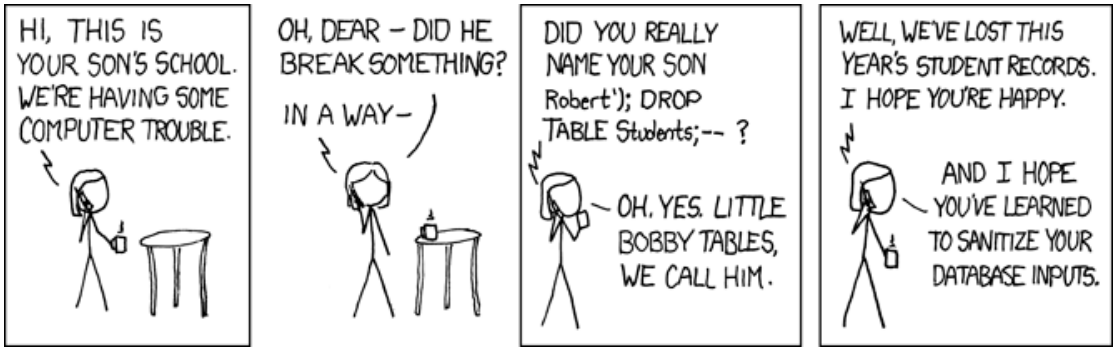
Tanulságok:

- SOSE fűzzünk össze programból SQL-t (semmilyen módszerrel), mert azzal kitesszük a kódunkat SQL Injection alapú támadásnak.
- az adatbázis állítsa össze a végleges SQL-t SQL paraméterek alapján, mert ilyenkor biztosított, hogy a paraméter értékek nem fognak SQL-ként értelmeződni (hiába írunk be SQL-t). Használjunk paraméterezett SQL-t vagy tárolt eljárást.
- használjunk adatbázis kényszereket, pl. a véletlen törlés ellen is véd
- konfiguráljunk adatbázisban felhasználókat különböző jogosultságokkal, a programunk connection string-jében megadott felhasználó csak a működéshez szükséges minimális jogokkal rendelkezzen. A mi esetünkben nem adtunk meg felhasználót, a windows-os felhasználóként fogunk csatlakozni.

4. Hívjuk meg az eredeti (vagyis a bizonságos, SQL paramétereket használó) beszúró függvényt a „speciális” paraméterezésekkel, hogy lássuk, működik-e a védelem:

```
InsertShipper("Super Shipper", "49-98562'); DELETE FROM Shippers;--");
InsertShipper("XXX');DELETE FROM Shippers;--", "49-98562");
```

Az elsőnél nem férünk bele a méretkorlátba, a második lefut, de csak egy „furcsa” nevű szállító került be. A paraméter értéke tényleg értékként értelmeződött nem pedig SQL-ként. Nem úgy mint itt:



Feladat 6 – Törlés

1. Írjunk egy új függvényt, ami kitöröl egy adott szállítót.

```
private static void DeleteShipper(int shipperID)
 {
     using (SqlConnection conn = new SqlConnection(CONN_STRING))
     using (SqlCommand command = new SqlCommand(
         "DELETE FROM Shippers WHERE ShipperID = @ShipperID",
         conn))
     {
         command.Parameters.AddWithValue("@ShipperID", shipperID);
         conn.Open();
         int affectedRows = command.ExecuteNonQuery();
         Console.WriteLine($"{affectedRows} row(s) affected");
     }
 }
```

5. Hívjuk meg új függvényünket a Main függvényből, pl. 1-essel, mint paraméterrel
6. Próbáljuk ki az alkalmazást. Valószínűleg kivételt kapunk, ugyanis van hivatkozás (idegen kulcs kényszerrel) az adott rekordra.
7. SSOE-ből nézzük ki az azonosítóját egy olyan szállítónak, amit mi vettünk fel. Adjuk át ezt az azonosítót a törlő függvénynek – ezt már ki tudja törölni, hiszen nincs rá hivatkozás.

Látható, hogy a törlés igen kockázatos és kiszámíthatatlan művelet az idegen kulcs kényszerek miatt. Néhány módszer a törlés kezelésére:

- kaszkád törlés – az idegen kulcs kényszeren beállítható, hogy a hivatkozott rekord törlésekor a hivatkozó rekord is törlődjön. Gyakran ez oda vezet, hogy minden idegen kulcs kényszerünk ilyen lesz, és egy (véletlen) törléssel végigtörölhetjük akár a teljes adatbázist, azaz nehezen jósolható a törlés hatása.
- hivatkozás NULL-ozása – az idegen kulcs kényszeren beállítható, hogy a hivatkozott rekord törlésekor a hivatkozó rekord idegen kulcs mezője NULL értékű legyen. Csak akkor alkalmazható, ha a modellünkben az adott idegen kulcs mező NULL-ozható.

- logikai törlés – törlés művelet helyett csak egy flag oszlopot (pl. Törölve) állítunk be. Előnye, hogy nem kell az idegen kulcs kényszerekkel foglalkoznunk, a törölt adat rendelkezésre áll, ha szükség lenne rá (pl. undelete művelet). Ám a működés bonyolódik, mert foglalkozni kell azzal, hogy hogyan és mikor szűrjük a törölt rekordokat (pl. hogy ne jelenjenek meg a felületen, statisztikákban), vagy hogyan kezeljük, ha egy nem törölt rekord töröltre hivatkozik.

Kitekintés

A fenti ADO.NET alapműveleteket ebben az itt látott alapformában ritkán használják a gyengén típusosság (egy rekord adatait beolvasni egy osztály property-jeibe igen körülményes, cast-olni kell, stb.) és a string-be kódolt SQL miatt – még akkor is, ha ez a hozzáférés adja a legjobb teljesítményt. Az előbbire megoldást jelenthetnek a különböző ADO.NET-et kiegészítő komponensek, pl.

- Dapper (https://github.com/StackExchange/dapper-dot-net)
- PetaPoco (http://www.toptensoftware.com/petapoco)

Ezek a megoldások egy minimális teljesítményveszteségért cserébe nagyobb kényelmet kínálnak.

Mindkét problémára megoldást jelentenek az ORM (Object-Relational-Mapping) rendszerek, cserébe ezek nagyobb overhead-del járnak, mint az előbb említett megoldások. Az ORM-ek leképezést alakítanak ki az adatbázis és az OO osztályaink között, és ennek a leképezésnek a segítségével egyszerűsítik az adatbázis műveleteket. Az osztályainkon végzett, típusos kóddal leírt műveleteinket automatikusan átfordítják a megfelelő adatbázis műveletekre, így a memóriabeli objektummodellünket szinkronban tartják az adatbázissal. Az ORM-ek ebből következően kapcsolat nélküli modellt használnak. Ismertebb .NET-es ORM-ek:

- ADO.NET DataSet – elsőgenerációs ORM, ma már ritkán használjuk.
- LINQ-to-SQL (elavult)
- Entity Framework – a jelenleg elsődlegesen használt .NET ORM (open source) .NET Framework alatt
- Entity Framework Core (EF Core) – a jelenleg elsődlegesen használt .NET ORM (open source) .NET Core alatt
- NHibernate – a java-s Hibernate .NET-es portja (open source)

Függelék – Adatbázisok, SQL nyelv alapok

Adatbázis kezelő rendszerek definíciója sokszor nem egyszerű feladat. Egy lehetséges definíció a következő:

Adatbázis definíció

Logikailag összefüggő adatok rendezett gyűjteménye, amely úgy van rendezve, hogy könnyű legyen az adatok tárolása, módosítása, illetve könnyen tudjunk keresni az adatokban. Összefüggő adatok: olyan adatok, amely az adott felhasználási terület szempontjából lényeges adatokat tartalmaz

pl. foci csapatok esetében a következőket szeretnénk tárolni (entitás és a hozzá tartozó tulajdonságok):

csapat: név, cím, bajnoki besorolás

játékos: név, cím, klub, fizetés, szerződés lejárta

meccs: résztvevő csapatok, időpont, eredmény, játékvezető, gólszerzők

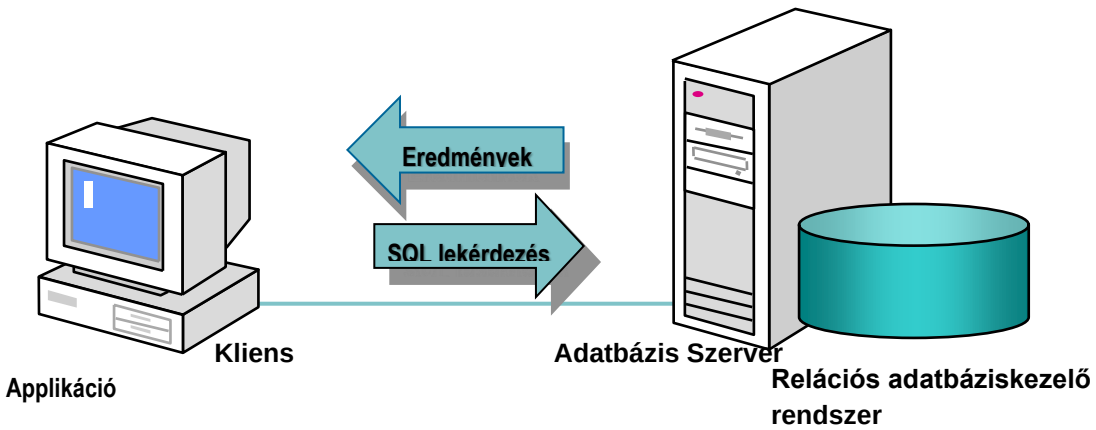
Adatbáziskezelők feladatai

Az adatbázis kezelő rendszerek a következő feladatok ellátását könnyítik meg számunkra:

- Adattárolás
- Adatstruktúrák fenntartása
- Több felhasználó egyidejű kiszolgálása
- Biztonság
- Adatok kinyerése és manipulálása
- Adatok beszúrása és betöltése
- Indexelés a gyors eléréshez
- Konzisztencia biztosítása
- Mentés-visszaállítás biztosítása

Architektúra

A mérés szempontjából a klasszikus kliens-szerver architektúra ismerete elégséges. Az adatbázis szerver egy külön alkalmazás keretein belül fut (külön processz – illetve processzek). Ez lehet egy gépen belül is, de általában külön erre a célra fenntartott gépen szokták működtetni.



1. ábra Architektúra

A működést tekintve a kliens kéréseket (szabványos SQL nyelven megfogalmazva pl. „SELECT név, cím FROM csapat”) küld a szerver felé, amire a szerver válaszol. A kommunikációra szabványos megoldásokat használnak. Napjaink elterjedt adatbázis kezelős rendszerei: Oracle ,MSSQL 2005, MySql, stb..

Egyed-Kapcsolat modell

Példa: Tegyük fel, hogy egy *könyvtár kölcsönzési nyilvántartását* szeretnénk adatbázissal megoldani. Ehhez nyilvántartást kell vezetni

- a könyvekről,
- az olvasókról,
- a kikölcsönzési és visszahozási időpontokról.

A modell megalkotásához néhány alapfogalmat meg kell ismernünk.

Egyednek vagy ***entitás***nak nevezünk egy, a valós világban létező dolgot, amit tulajdonságokkal akarunk leírni. Esetünkben egyed lehet egy könyv a könyvtárban, illetve egy adott olvasó. Általánosított fogalmakat használva beszélhetünk "könyv" egyedről és "olvasó" egyedről is.

Tulajdonságnak vagy ***attribútum***nak nevezzük az egyed egy jellemzőjét. Például a könyv, mint egyed legfontosabb tulajdonságai a címe, és a szerző neve.

Az attribútumokat úgy célszerű megválasztani, hogy azok a felhasználási terület szempontjából érdemi információt hordozzanak és segítségükkel az egyedek megkülönböztethetők legyenek. Mivel adott szerző adott című könyve több kiadásban is megjelenhet, sőt adott kiadásból is több példány lehet a könyvtárban, így minden könyvhöz egy egyedi azonosítót, *könyvszámot* (könyvtári számot) célszerű felvenni. Ekkor a "könyv" egyed tulajdonságai: *könyvszám, szerző, cím*. (További tulajdonságoktól, mint kiadó, kiadási év, stb. esetünkben eltekintünk.) Hasonló meggondolások alapján az "olvasó" egyedhez *olvasószám, név, lakcím* tulajdonságokat rendelhetünk.
Egy egyed attribútumainak azt a minimális részhalmazát, amely egyértelműen meghatározza az egyedet, ***kulcs***nak nevezzük és *aláhúzás*sal jelöljük. Esetünkben a „könyv” egyed kulcsa a *könyvszám*, az „olvasó” egyedé az olvasószám.

Könyvtári nyilvántartásunk azonban ezzel még nincs kész. A "könyv" és "olvasó" egyedek között ugyanis egy sajátos *kapcsolat* léphet fel, amelyet *kölcsönzés*nek nevezünk. Ezen kapcsolathoz a kivétel és visszahozás időpontját rendelhetjük tulajdonságként.

A valós világ jelenségeit egyedekkel, tulajdonságokkal és kapcsolatokkal leíró modellt *egyed-kapcsolat modell*nek, az ezt ábrázoló diagramot *egyed-kapcsolat diagram*nak nevezik. (Rövidítve az *E-K modell* és *E-K diagram*, illetve az angol *entity-relationship model* elnevezés alapján az *E-R modell* és az *E-R diagram* elnevezések használatosak.)

Relációs adatmodell fogalma

A gyakorlatban szinte kizárólag relációs adatbázis kezelőket használnak. Relációs modellekben táblákat definiálnak.

Az adattábla (vagy egyszerűen csak *tábla*) sorokból és oszlopokból áll. Egy sorát *rekord*nak nevezzük, amely annyi *mező*ből áll, ahány oszlopa van a táblának. A táblára példát a következő fejezet 2. ábrája mutat.
*Relációséma*nak nevezünk egy attribútumhalmazt, amelyhez azonosító nevet rendelünk.
A tábla ***minden sora különböző*****, és a sorokra *semmilyen rendezettséget nem tételez fel*.**

Redundáns adatok

Tekintsük egy vállalat dolgozóit nyilvántartó

DOLGOZÓ (Név, Adószám, Cím, Osztálykód, Osztálynév, VezAdószám)

táblát, ahol *VezAdószám* a vállalati osztály vezetőjének adószámát jelenti. A megfelelő tábla a 2. ábrán látható.

Előny: egyetlen táblában a dolgozók és osztályok adatai is nyilvántartva.

Hátrány: redundancia, mivel *Osztálynév, VezAdószám* több helyen szerepel.

Név	*Adószám*	*Cím*	*Osztálykód*	*Osztálynév*	*VezAdószám*
Kovács	1111	Pécs, Vár u.5.	2	Tervezési	8888
Tóth	2222	Tata, Tó u.2.	1	Munkaügyi	3333
Kovács	3333	Vác, Róka u.1.	1	Munkaügyi	3333
Török	8888	Pécs, Sas u.8.	2	Tervezési	8888
Kiss	4444	Pápa, Kő tér 2.	3	Kutatási	4444
Takács	5555	Győr, Pap u. 7.	1	Munkaügyi	3333
Fekete	6666	Pécs, Hegy u.5.	3	Kutatási	4444
Nagy	7777	Pécs, Cső u.25.	3	Kutatási	4444

2. ábra. Dolgozók nyilvántartását tartalmazó redundáns tábla

A redundancia aktualizálási anomáliákat okozhat:

(i) Módosítás esetén:

- Ha egy osztály neve vagy vezetője megváltozik, több helyen kell a módosítást elvégezni, ami hibákhoz vezethet.

(ii) Új felvétel esetén:

- Új dolgozó felvételénél előfordulhat, hogy az osztálynevet máshogy adják meg (például *Tervezési* helyett *tervezési* vagy *Tervező*).
- Ha új osztály létesül, amelynek még nincsenek alkalmazottai, akkor ezt csak úgy tudjuk felvenni, ha a *név, adószám, cím* mezőkhöz NULL értéket veszünk. Később, ha lesznek alkalmazottak, ez a rekord fölöslegessé válik.

(iii) Törlés esetén:

- Ha egy osztály valamennyi dolgozóját töröljük, akkor az osztályra vonatkozó információk is elvesznek.

Megoldás: a tábla felbontása két táblára (dekompozíció):

DOLGOZÓ (Név, Adószám, Cím, Osztálykód)

OSZTÁLY (Osztálykód, Osztálynév, VezAdószám)

A szétválasztott táblák a 3. ábrán láthatók.

DOLGOZÓ tábla minta adatokkal:

Név	*Adószám*	*Cím*	*Osztálykód*
Kovács	1111	Pécs, Vár u.5.	2
Tóth	2222	Tata, Tó u.2.	1
Kovács	3333	Vác, Róka u.1.	1
Török	8888	Pécs, Sas u.8.	2
Kiss	4444	Pápa, Kő tér 2.	3
Takács	5555	Győr, Pap u. 7.	1
Fekete	6666	Pécs, Hegy u.5.	3
Nagy	7777	Pécs, Cső u.25.	3

OSZTÁLY tábla minta adatokkal:

Osztálykód	*Osztálynév*	*VezAdószám*
1	Munkaügyi	3333
2	Tervezési	8888
3	Kutatási	4444

3. ábra. Redundancia megszüntetése a tábla felbontásával

Megjegyzés: Ha helyesen felírt E-K modellből indulunk ki, amely a *Dolgozó* és *Osztály* entitások között két kapcsolatot (*dolgozik* és *vezeti*) tartalmaz, akkor eleve a fenti két táblához jutunk.

A felmerülő kérdések
- mikor van redundancia egy táblában,
- hogyan kell ezt a tábla felbontásával megszüntetni.

(válaszok: Adatbázisok c. tárgy keretein belül):

Az SQL nyelv

Az SQL nyelv egy szabványos nyelv, amelynek segítségével kezelhetjük adatainkat.
Pl: tekintsük a 3. ábra második tábláját (T). Szeretnénk megtudni az 1-es Osztálykódhoz tartozó Osztálynév, VezAdószám értékeket, akkor a következőkezt írhatjuk:

```
SELECT Osztálynév, VezAdószám FROM T WHERE Osztálykód=1 OR Osztálykód=2
```

, aminek kimenete esetünkben két rekord:

Osztálynév	*VezAdószám*
Munkaügyi	3333
Tervezési	8888

Az SQL utasításait két fő csoportba szokták sorolni:
- DDL (= Data Definition Language): adatstruktúra definiáló utasítások.
- DML (= Data Manipulation Language): adatokon műveletet végző utasítások.

Jelen anyagban - az RDBMS fő feladatai alapján - az alábbi csoportokban tárgyaljuk az SQL utasításokat:

- adatbázisséma definiálása (DDL),
- adatok módosítása (DML),
- lekérdezési lehetőségek (DML).

Táblák definiálása (DDL)

Táblák létrehozására a CREATE TABLE utasítás szolgál, amely egy üres táblát hoz létre. Az attribútumok definiálása mellett a kulcsok és külső kulcsok (ennek segítségével tudunk táblákat összekapcsolni) megadására is lehetőséget nyújt:

CREATE TABLE táblanév
(oszlopnév adattípus [feltétel],
... ...,
oszlopnév adattípus [feltétel]
[, táblaFeltételek]
);

Az adattípusok (rendszerenként eltérők lehetnek):

CHAR(n)	n hosszúságú karaktersorozat
VARCHAR(n)	legfeljebb n hosszúságú karaktersorozat
INT	egész szám (röviden INT)
REAL	valós (lebegőpontos) szám, másnéven FLOAT
DECIMAL(n[,d])	n jegyű decimális szám, ebből d tizedesjegy
DATETIME	dátum

…

Megjegyzés

Feltételek (egy adott oszlopra vonatkoznak):

PRIMARY KEY: elsődleges kulcs

UNIQUE: kulcs

REFERENCES tábla(oszlop) [ON-feltételek]: külső kulcs

Táblafeltételek (az egész táblára vonatkoznak):

PRIMARY KEY (oszloplista): elsődleges kulcs – egyértelműen megkülönböztetik a rekordokat

UNIQUE (oszloplista): kulcs

FOREIGN KEY (oszloplista) REFERENCES tábla(oszloplista) [ON-feltételek]: külső kulcs

Ha a (külső) kulcs több oszlopból áll, akkor csak táblafeltétel formájában adható meg.

A PRIMARY KEY (elsődleges kulcs) és UNIQUE (kulcs) közötti különbségek:

- Egy táblában csak egy elsődleges kulcs, de tetszőleges számú további kulcs lehet.
- Külső kulccsal csak a másik tábla elsődleges kulcsára lehet hivatkozni.
- Egyes DBMS-ek az elsődleges kulcshoz automatikusan indexet hoznak létre.)

Példa. Hozzuk létre az
OSZTÁLY (osztálykód, osztálynév, vezAdószám)
DOLGOZÓ (adószám, név, lakcím, *osztálykód*)
relációsémákat SQL-ben:

```
CREATE TABLE Osztály
  ( osztálykód   CHAR(3)  PRIMARY KEY,
    osztálynév   CHAR(20),
    vezAdószám   DECIMAL(10)
  );
CREATE TABLE Dolgozó
  ( adószám    DECIMAL(10)  PRIMARY KEY,
    név        CHAR(30),
    lakcím     CHAR(40)  DEFAULT 'ismeretlen',
    osztálykód CHAR(3)  REFERENCES Osztály(osztálykód)
  );
```

A DOLGOZÓ sémát így is lehetne definiálni:

```
CREATE TABLE Dolgozó
  ( adószám    DECIMAL(10),
    név        CHAR(30),
    lakcím     CHAR(40),
    osztálykód CHAR(3),
    PRIMARY KEY (adószám),
    FOREIGN KEY (osztálykód) REFERENCES Osztály(osztálykód)
  );
```

A tábla módosításakor a definiált kulcsfeltételek automatikusan ellenőrzésre kerülnek. PRIMARY KEY és UNIQUE esetén ez azt jelenti, hogy a **rendszer nem enged** olyan módosítást illetve új sor felvételét, amely egy már meglévő kulccsal ütközne.

Adattábla módosítása (DML)

A táblába *új sor felvétele* az

INSERT INTO táblanév [(oszloplista)] VALUES (értéklista);

utasítással történik. Ha *oszloplista* nem szerepel, akkor valamennyi oszlop értéket kap a CREATE TABLE-ben megadott sorrendben. Egyébként, az oszlopnév-listában nem szereplő mezők NULL értéket kapnak.

Példák:

```
INSERT INTO Dolgozó (név, adószám)
   VALUES ("Tóth Aladár", 1111);
INSERT INTO Dolgozó
   VALUES (1111, "Tóth Aladár", , "12");
```

A táblába adatokat tölthetünk át másik táblából is, ha a VALUES(értéklista) helyére egy alkérdést írunk (lásd az *Alkérdések* fejezetben).

Sor(ok) módosítása az

UPDATE táblanév
SET oszlop = kifejezés, ..., oszlop = kifejezés
[WHERE feltétel];

utasítással történik. Az értékadás minden olyan soron végrehajtódik, amely eleget tesz a WHERE feltételnek. Ha WHERE feltétel nem szerepel, akkor az értékadás az összes sorra megtörténik.

Példák:

```
UPDATE Dolgozó
   SET lakcím = "Szeged, Rózsa u. 5."
   WHERE név = "Kovács József";
UPDATE Dolgozó
   SET osztálykód = "003"
   WHERE osztálykód = "012";
```

Sor(ok) törlése a

DELETE FROM táblanév
[WHERE feltétel];

utasítással lehetséges. Hatására azok a sorok törlődnek, amelyek eleget tesznek a WHERE feltételnek. Ha a WHERE feltételt elhagyjuk, akkor az összes sor törlődik (de a séma megmarad).

Példa:

```
DELETE FROM Dolgozó
   WHERE név = "Kovács József";
DELETE FROM Osztály;
```

Tekintsük az alábbi utasításpárt:

```
INSERT INTO Dolgozó (név, adószám)
   VALUES ("Tóth Aladár",4321);
DELETE FROM Dolgozó WHERE adószám = 4321;
```

Ha a táblában korábban már volt egy 4321 adószámú sor, akkor a fenti utasításpár azt is kitörli. Általában, ha egy tábla két azonos sort tartalmaz, DELETE utasítással nem tudjuk csak az egyiket kitörölni. Ha ugyanis a WHERE feltétel az egyikre igaz, akkor szükségképpen a másikra is igaz. A PRIMARY KEY feltétellel az ilyen anomáliák megelőzhetők.

Lekérdezés

Lekérdezésre a SELECT utasítás szolgál, amely egy vagy több adattáblából egy *eredménytáblát* állít elő. Az eredménytábla a képernyőn listázásra kerül, vagy más módon használható fel. (Egyetlen SELECT akár egy komplex felhasználói programot helyettesíthet!)

A SELECT utasítás alapváltozata:

SELECT [DISTINCT] oszloplista
FROM táblanévlista
[WHERE feltétel];

```
SELECT Osztálynév, VezAdószám FROM T WHERE Osztálykód=1
```

(Megj: a több táblából való lekérdezés az Adatbázisok című tárgyban)
Ha *oszloplista* helyére * karaktert írunk, ez valamennyi oszlop felsorolásával egyenértékű. A SELECT legegyszerűbb változatával adattábla listázását érhetjük el:

```
SELECT * FROM T;
```