Fóti Marcell – Turóczy Attila

Adatkezelés

otthon és a felhőben

A **Microsoft SQL Server 2012**
és a **Microsoft SQL Azure** alkalmazása

Microsoft

J.O.S.

Készült a devPortal.hu támogatásával

Fóti Marcell – Turóczy Attila

Adatkezelés otthon és a felhőben

A Microsoft SQL Server 2012 és a Microsoft SQL Azure alkalmazása

JEDLIK OKTATÁSI STÚDIÓ
Budapest, 2012

Borító: Varga Tamás

Anyanyelvi lektor: Dr. Bonhardtné Hoffmann Ildikó

Kiadó: Jedlik Oktatási Stúdió Kft.
1215 Budapest, Ív u. 8-12.
Internet: http://www.jos.hu
E-mail: jos@jos.hu
Felelős kiadó: a Jedlik Oktatási Stúdió Kft. ügyvezetője

Nyomta: LAGrade Kft.
Felelős vezető: Szutter Lénárd

ISBN: 978-615-5012-18-1
Raktári szám: JO-0341

Bevezető

2012 tavaszán, majdnem fél évszázaddal az SQL-nyelv kifejlesztése után elérkezettnek láttam az időt, hogy írjak egy könyvet a Transact SQL nyelvről. Hogy miért pont most? És egyáltalán minek, ha a világ telis-tele van SQL-könyvekkel? Azért, mert angol nyelvű szakirodalom van bőven, talán még naprakész is akad, de magyarázó leírásokkal tűzdelt, a legújabb fejlesztéseket is figyelembe vevő magyar nyelvű könyv nincs a piacon. Emellett jó apropót kínált, hogy ha most írok könyvet, elcsípem az SQL Serverek új hullámát, az SQL 2012-t, és olyan ultramodern dolgokról is írhatok, amelyek nélkül ugyan van élet, de milyen? No meg a lustaság is vezérelt, hogy ha most ezt megírom, a következő verzióig, azaz legalább három évig nem lesz dolgom a könyv frissítésével.

Felsorolván a motivációimat, akár bele is kezdhetünk a lényegi részbe. Minden ilyen könyv kötelezően tartalmaz valamiféle történeti áttekintést. Mivel az SQL-nyelv és a Microsoft SQL Server története eléggé regényes, én sem hagyom ki ezt a lehetőséget.

Kezdjük talán azzal, hogy ha valaki egy külföldi előadást meghallgat az SQL Serverről, észreveheti, hogy a beszélők – különösen a régi motorosok – úgy ejtik ki az SQL-t, hogy „szíkúel” vagy „szíkvel”. No már most ha valaki tanulta annó az angol ábécét, vagy ha nem is tanulta, de ismeri az ábécés gyermeknótát, még ha eddig el is kerülte esetleg a figyelmét, most biztosan felrémlik előtte, hogy a „szí” az bizony nem az S, hanem a C betű. Akkor miről is beszélnek ezek? CQL-ről? Az meg mi?

Nos, a CQL az nem más, mint SEQUEL, avagy Structured English Query Language[1], azaz struktúrált angol lekérdezési nyelv. A 70-es években az IBM kutatói az SQL-nyelvet nem programozási nyelvként hozták létre. A nyelv eredeti „célcsoportja” ugyanis az adatbázis-alkalmazások felhasználói vagy legalábbis felügyelői voltak, célja pedig nem más, mint hogy az alkalmazásokból programozás nélkül, pusztán egy természetes nyelv használatával okosan elő lehessen állítani különböző halmazokat. A nemes cél igen gyorsan megbukott, mert a felhasználók két területen is gyarlónak bizonyultak: egyfelől nem lehet elvárni egy humanoidtól, hogy ismerje az adatbázis pontos szerkezetét, amely nélkül a lekérdezés eleve kudarcra van ítélve, másfelől egy átlagos ember nem képes kötött szintaxist használva kifejezni magát. Minden SQL-parancs mind a mai napig egy-egy hibátlan angol nyelvű mondat ugyan, de csak egyetlen helyes változat a lehetséges kismillió kombináció közül. Tehát mind az adatbázis szerkezetét, mind a lekérdezési nyelvet tanulni kell. Ezért olvasod most ezt a könyvet.

A nyelvet így átkeresztelték SQL-re, amiben persze szerepet játszott egy jó kis szabadalmi per is, mivel a SEQEL rövidítésre benyújtotta igényét egy brit repülőgyár is. A szabadalmi csetepaték nem mai keletűek, már eleink is térdig gázoltak bennük.

[1] http://en.wikipedia.org/wiki/SQL

Létezik egy folyamatosan fejlődő szabvány magára az SQL-nyelvre, amelyet a nagy tekintélyű ANSI szabványszervezet tart karban. Ez az SQL-nyelvet több részre bontja:

- Data Definition Language, a táblák és egyéb objektumok létrehozására, módosítására és törlésére, tehát CREATE, ALTER, DROP
- Data Query Language, a lekérdezőnyelv, vagyis a jó öreg SELECT
- Data Manipulation Language, azaz INSERT, UPDATE, DELETE
- Data Control Language, a jogosultságállítgatás nyelve, GRANT, REVOKE, DENY

A piacon számtalan SQL-alapú adatbáziskezelő van, gyors felsorolásképpen csak néhány név: Oracle, MySQL, DB2, Informix, Sybase és persze a Microsoft SQL Server. Az ANSI SQL-szabvány tükrében azt gondolhatnánk, hogy az imént felsorolt rendszerek SQL-nyelve azonos. Nos, ez egyáltalán nincs így. A jelenség oka, hogy az ANSI SQL követőüzemmódban dolgozik, a fejlődését pedig a piaci szereplők diktálják, de olyan iramban, hogy a szabványtestület csak kapkodja a fejét. A legfrissebb szabvány az ANSI SQL 2008, de hol van már 2008?

Emiatt sajnos a szabványban megjelenő minden „új” elem már rég megvalósult az egyes adatbáziskezelőkben, mégpedig úgy, ahogy az egyes gyártók azt jónak találták. Ezért nincs két egyforma trigger, ezért léteznek egyedi, eltérő JOIN szintaxisok. Sőt, bizonyos alapvető területeket a szabvány egész egyszerűen nem vesz figyelembe, mintha nem is léteznének. A triggereket például csak 1999-ben „legalizálták”, de addigra mindegyik gyártónak már a sokadik verziójú triggermegoldása létezett párhuzamosan. Tudtommal nem is változtatta meg egyik sem a maga triggerimplementációját emiatt az apróság miatt.

Ezzel el is jutottunk a Transact SQL-nyelvhez, ami nem más, mint a szabványos ANSI SQL-nyelv kiegészítése mindazzal, amit a Microsoft, illetve korábban a Sybase jónak látott.

Mi?

Igen, jól olvastad. A regényes történelem feltárja előttünk, hogy a Microsoft SQL Server nem más, mint a SyBase SQL Server „forkolása”. A Microsoftnak ez a terméke is felvásárlás útján került a kínálatba. A Microsoft SQL Server legelső verziója, a 4.2 megegyezett a SyBase SQL Server 4.2-es verziójával. A Microsoft ugyanis megvásárolta a SyBase forráskódját, amit a lüke SyBase a következő korlátozásokkal bocsátott a Microsoft rendelkezésére: rendben van, uraim, de Önök a saját példányukat csak és kizárólag Windows platformon használhatják. Hát ez pont elegendő volt a Microsoftnak ahhoz, hogy SQL Serverével a semmibe nyomja a SyBase-t, és megszorongassa akár az Oracle-t, akár a DB2-t.

(Kitérő: Nekem volt „szerencsém” dolgozni a Microsoft SQL Server 4.21A verzióval. Pestiesen szólva – „netuddmeg”. Nem volt benne szinte semmi a maiakhoz képest! Egy nyomorult növekvő azonosító megszerzéséhez SELECT MAX(*) FROM TÁBLA lekérdezést kellett írni, amivel az ember persze beborította a párhuzamos végrehajtásnak a lehetőségét is.)

Ennyit a dicső múltról, most jöjjön a jelen és a jövő!

Tartalomjegyzék

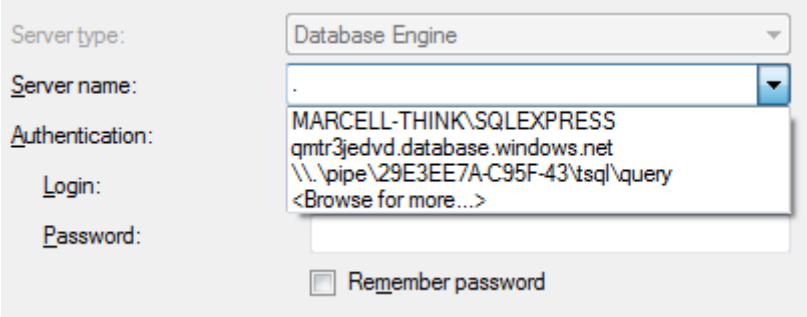
1 Ismerkedés az SQL Server Management Studióval

1.1 Kapcsolódás SQL Serverhez, adatbázishoz

Első fejezetünkben végigegerészünk az SQL Server Management Studio felületén és lehetőségein, végigpróbálva mindazt, ami fontos, kihagyva azokat az elemeket, amelyek túl sok haszonnal nem kecsegtetnek. Egy SQL-könyvnek túlnyomó részben SQL-parancsokat, scripteket kell tartalmaznia, ezért az egértologatást lerendezzük az első fejezetben. A továbbiakban már mindent SQL-paranccsal fogunk csinálni. Most azonban még kell a kattintgatás, hiszen még be sem léptünk az SSMS néven rövidített legfőbb rendszerfelügyeleti eszközbe, az SQL Server Management Studióba.

Tegyük fel, hogy a Kedves Olvasók rendelkeznek valamilyen SQL Server 2012 példánnyal, amiben rendszergazdai jogosultságot élveznek (különben a parancsok fele nem működne nekik), amire vagy úgy tettek szert, hogy ők egy vállalati SQL Server felelős szakemberei, vagy telepítettek maguknak egy 180 napos ingyenes próbaverziót, esetleg leszedték és telepítették az ingyenes SQL Expresst. Miután mindenki tisztázta magában, melyik SQL Serveren dolgozhat korlátlanul, indítsa el az SSMS-t a Start menüből, illetve ha olyan nincs, akkor a Windows 8 „arcába” kezdje begépelni, hogy SQL, és máris megkapja a találati listában! (Ilyen triviális lépéseket nem szeretnék részletesen kifejteni, az az SQL-guru-aspiráns, aki nem tudja, mi az a Start menü, sürgősen forduljon szakemberhez!)

Első lépésként kapcsolódjunk egy SQL Serverhez! Az ábrán az én SSMS-em kapcsolati listája látható, amin szerepel egy-két speciális szervernév, amiket hamarosan röviden elmagyarázok. Normális esetben a Server name mezőben mindössze az adott számítógép neve szerepel. De nálam ez:



1. ábra - kapcsolódás SQL Serverekhez

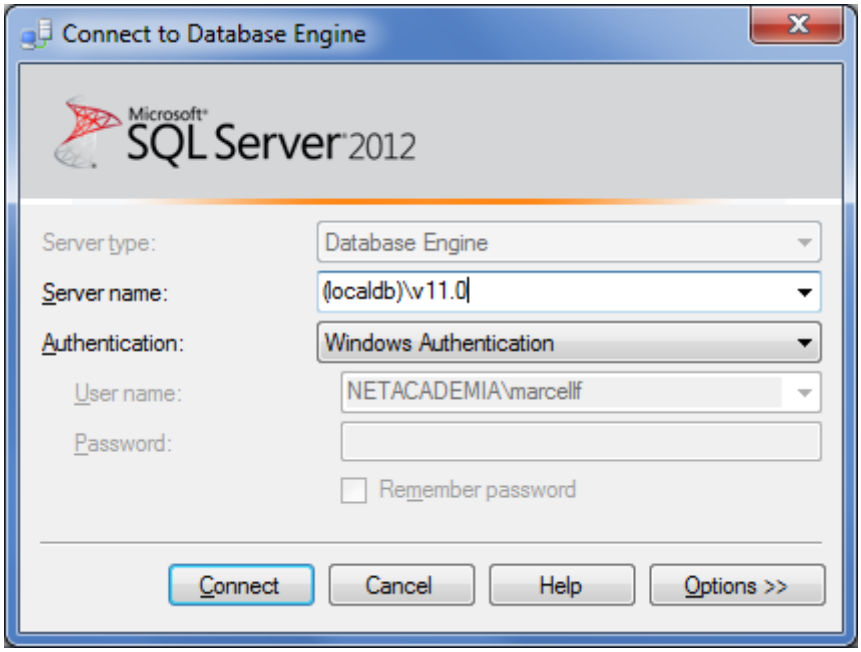
Soronként a következőket látjuk:

- Beírtam a mezőbe egy pontot. Ez mindig az aktuális, helyi SQL Server tőpéldányt jelenti, ezzel a helyben telepített elsődleges (nem nevesített) példányhoz lehet kapcsolódni. A „pont” trükköt akkor szoktam használni, ha fogalmam sincs a Windows számítógépnevéről. *(A ponttal egyenértékű a (local) is, ha valaki többet szeret gépelni…)*
- A listában a legfelső sor egy SQL Server nevesített példányhoz kapcsolódik, amelyik a laptopomon lakik. A neve: MARCELL-THINK\SQLEXPRESS. *(Ezt egyébként a Visual Studióm telepítette fel.)*
- A második a Windows Azure-ban található SQL Serverem becses neve. Bátran leírhatom ide, mert az Azure szervereket a jogosultságon felül szigorú IP-cím alapú hozzáférési szabályok

védik, lehet próbálkozni a hekkeléssel, ha valakinek úgy tetszik. A neve: qmtr3jedvd.database.windows.net

- A harmadik, furcsa nevű valami a Visual Studióban futtatott ASP.NET MVC webalkalmazásom AppData könyvtárából induló, User kontextusban futó adatbázis, aminek a nevét nem volt triviális összevadászni, de így most rá tudok kapcsolódni az SSMS-szel, hogy teljes körű hozzáférést kapjak a Visual Stúdio korlátozott lehetőségei helyett[2].

Az ábrán nem látszik, de van még egy kapcsolódási lehetőség, ugyanis az SQL Expresszel, vagy ahelyett lehet telepíteni a LocalDB nevű még kisebb adatbázist, ami annyira mini, hogy még szolgáltatást sem telepít, hanem ha rácsatlakozunk, elindul az sqlserver.exe processz, és ahhoz tudunk kapcsolódni. A LocalDB funkciójában megegyezik a legnagyobb kiépítésű, Enterprise változattal, mert az a célja, hogy a fejlesztőknek olyan szolgáltatáskészletet biztosítson telepítgetés nélkül, amilyen csak a nagy adatközpontokban van. A LocalDB-hez így lehet csatlakozni, és mivel ez az információ nem lelhető fel sehol a szakirodalomban (legalábbis 2012. április 19-én még nem, vagy nem találtam meg), ezzel az egy képernyőképpel nélkülözhetetlenné és felbecsülhetetlenné tettem ezt a könyvet:



2. ábra - csatlakozás a LocalDB-hez

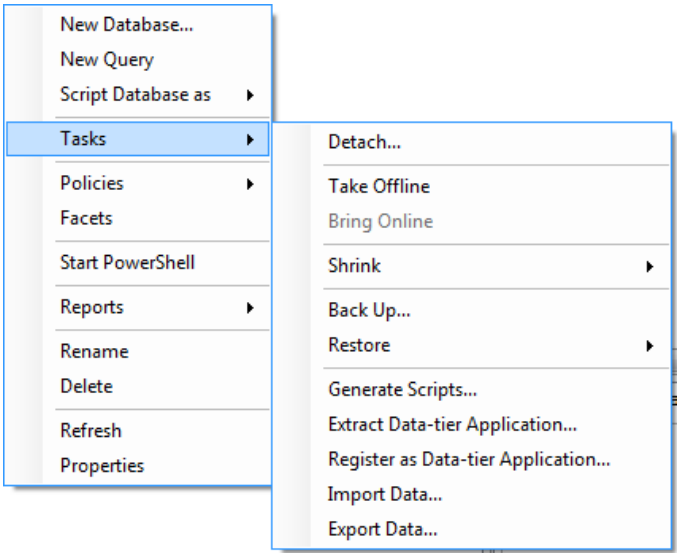
A trükk tehát a következő: az SQL Server neve helyére ezt kell írni: (localdb)\v11.0

Ha beírtuk az SQL Server nevét (vagy egy pontot, a helyi SQL Server elérésére), következik az autentikáció. Windowsos környezetben jó eséllyel Windows logint, felhőben nagy valószínűséggel SQL-logint fogunk használni. Ez utóbbi esetben az SQL Server maga kezeli a bejelentkezési neveket, jelszavakat.

[2] Ezt a mondatot is rágja az idő vasfoga. Az SQL Server Data Toolsszal a Visual Studio is teljes körű képességeket kap.

1.2 Mit csinál a „fa”?

Miután létrejött a kapcsolat és beléptünk, bal szélen egy Object Explorer névre hallgató faszerkezetben látjuk az SQL Serverünk vickeit-vackait. Vannak neki adatbázisai, biztonsági beállításai stb. Minket most kizárólag az adatbázis ág érdekel. Amit fontos tudnunk a fáról és a helyzetérzékeny menüről, az az, hogy minden szinten szinte minden van. Érdemes legalább egyszer jobb gombbal végigkattogni az egészen, mert minden objektumnál más és más, azon a ponton releváns menüpontok bukkannak fel. Hogy csak egyetlen példát mutassak *(a többi házi feladat)*, egy adatbázison például az alábbi tekintélyes mennyiségű funkció érhető el:



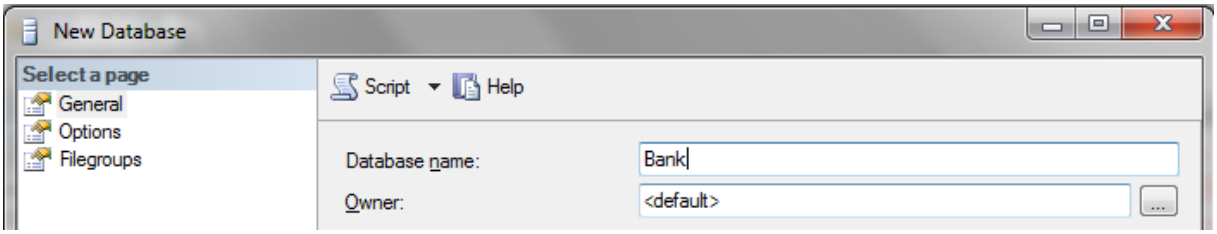
3. ábra - az adatbázis helyi menüje

Ezenfelül a fa egy SQL-tanítómester is egyben, mert bármit csinálunk, végül ő is csak SQL-scriptet futtat, amit az SQL Profilerrel elleshetünk tőle, és később azzal vagánykodhatunk, hogy egy paranccsal meg tudunk csinálni roppant összetettnek tűnő feladatokat.

1.3 Adatbázis, táblák, diagramok létrehozása

Jöhet az első érdemi lépés, az adatbázis létrehozása. A könyvben egy online bank adatbázisának létrehozását követhetjük végig, ezért az új adatbázis neve legyen Bank!

Mivel most még egerészős üzemmódban vagyunk, a bal oldali fában jobbkattintunk a Databases ágon, kiválasztjuk a New Database… menüpontot, és a felbukkanó ablakban megadjuk az adatbázisunk nevét, így:



4. ábra - új adatbázis létrehozása

Az adatbázist most alapértelmezett méretben, alapértelmezett helyen, alapértelmezett növekedési beállításokkal hozzuk létre. Ha majd egyszer – netalán – írok egy SQL Admin könyvet is, ott részletesen kitérek ezekre, vagy tudom javasolni a 40 órás SQL Admin tanfolyamot. Most azonban click oké!

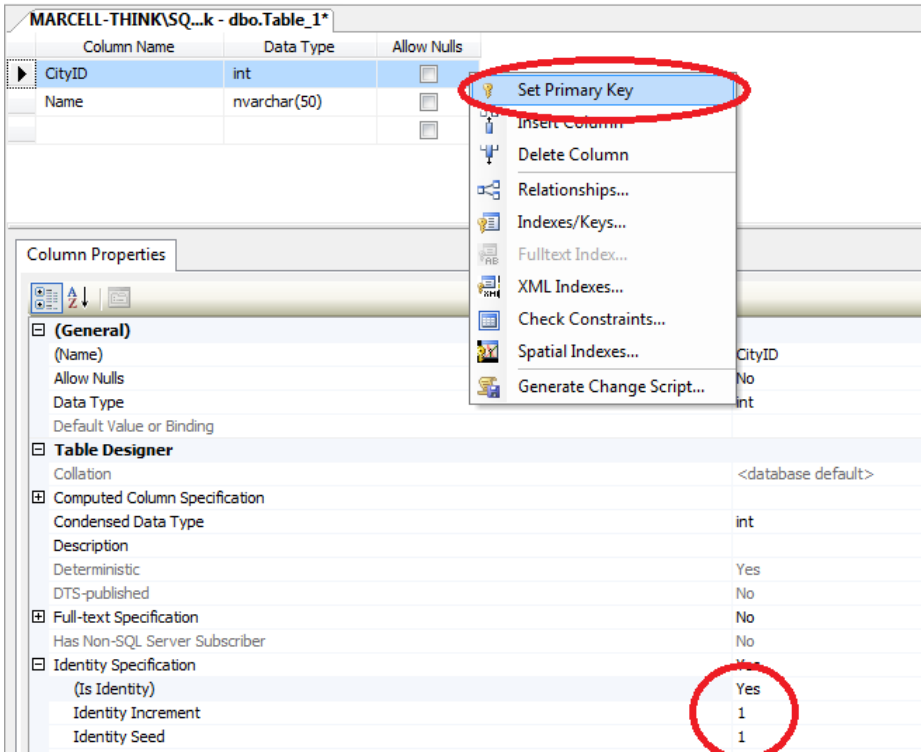
Mivel több adatbázist nem szándékozunk létrehozni, itt és most megadom a létrehozás SQL-parancsát is, el ne maradjon:

```
CREATE DATABASE Bank
```

(Kis-nagybetű: érdemes elköteleznünk magunkat egy adott írásmód mellett, mert a scriptjeink ugyan látszólag kis-nagybetű érzéketlenek, de ez csak azért van így, mert az örökölt adatbázis-beállításunk a nyelvekre vonatkozóan most épp ilyen (COLLATION). Ha azonban meg kell változtatnunk az adatbázis nyelviségét és „kézszenzitivitását", az összevissza írt scriptek azonnal fel fogják dobni a talpukat. Én az SQL-parancsok tekintetében a csupa nagybetűre szavazok, az objektumok neve pedig nálam nagybetűvel kezdődik.)

És már jöhet is a táblák létrehozása. Mivel még mindig egerészünk, első tábláinkat a Table Desingerrel fogjuk létrehozni. Ebből két változat is van, az egyszerűbbet úgy érjük el, hogy a fában kibontogatjuk újdonsült adatbázisunkat a Tables szintig, és ott jó erősen kattintunk a jobb gombbal, a kismenüből pedig kiválasztjuk a New table… menüpontot. Ennek hatására a jobb térfelet teljesen el fogja borítani az úgynevezett Table Designer. Készítsünk egy rendkívül egyszerű táblát, egyelőre anélkül, hogy elmerülnénk az adattípusok rejtelmei közé! Legyen az első táblánk a városok! Egyezzünk meg abban, hogy egy nemzetközi bank adatbázisát készítjük el, ennek megfelelően a tábla neve Cities lesz, és nem Varosok! *(Ez a döntés egyébként a későbbiekben kifizetődik, amikor majd valaki kliensprogramot készít az adatbázisunkhoz, és igénybe veszi az Entity Framework egyes szám-többes szám átnevezőkéjét az objektumokhoz, mert az nagyon szépen elboldogul az angol nyelvvel – és csak azzal.)*

A Cities tábla tartalmazzon egy automatikusan növekedő egyedi azonosítót és egy városnevet! Az alábbi ábrára próbáltam az összes szükséges kattintást rázsúfolni, mert be kell állítanunk az INT mezőn egy IDENTITY értéket az automatikus számláláshoz, valamint ki is kell jelölnünk ugyanezt a mezőt elsődleges kulcsnak:

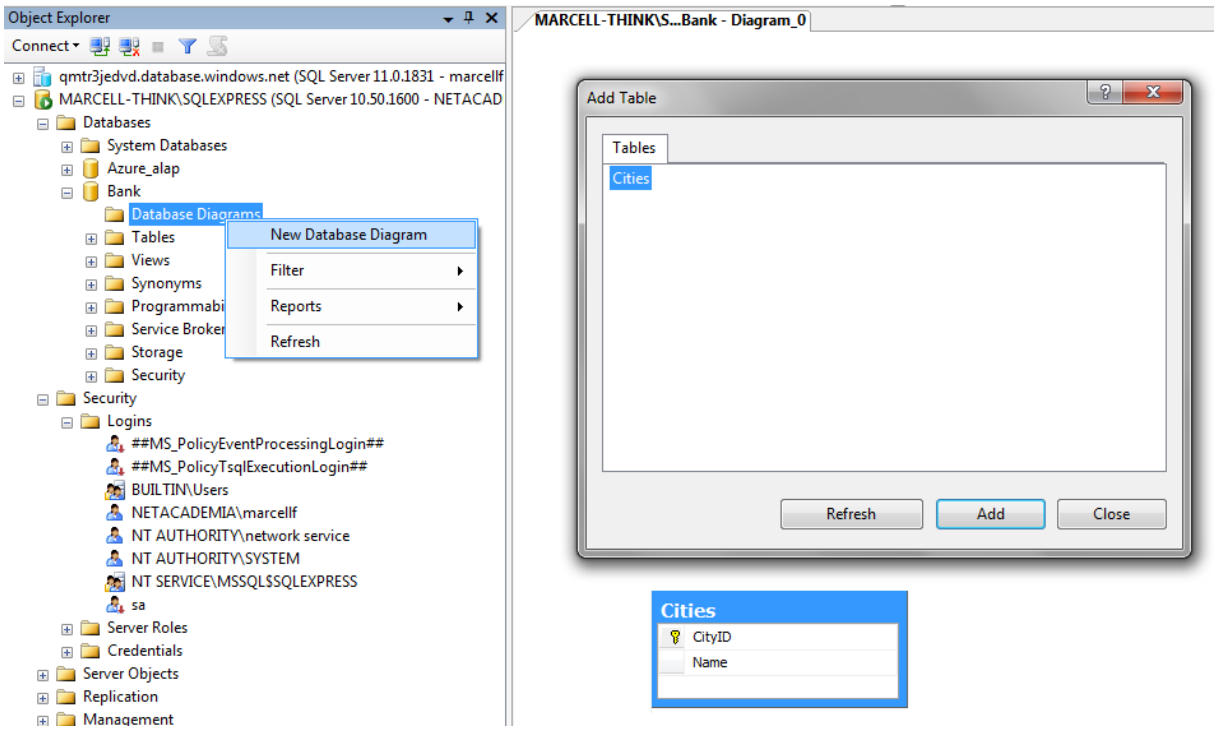


5. ábra - egyes számú Table Designer

Figyeljük meg, hogy mindkét mező esetén kitiltottam a NULL értékeket! A CityID-nél ez remélem egyértelmű, a Name mezőnél pedig azért, mert névtelen város nincsen. Ha nem tudjuk egy város nevét, véletlenül se hozzunk létre olyan rekordot, ami üres nevet tartalmaz! Ha végeztünk, keressük meg az eszközsoron a mentés ikont *(az idősebb nemzedék kedvéért: floppylemez, a fiatalabbak meg keressék a feje tetejére állított mosogatógépet)*, katt oda, adjuk a táblának a Cities nevet, és készen vagyunk. Új táblánk szépen megjelenik a fában bal oldalon, mindenféle frissítgetés nélkül. *(Ez csak természetes, nem? Ha majd scriptekkel dolgozunk, egyáltalán nem lesz természetes. Sem a fa, sem az IntelliSense nem fogja automatikusan észrevenni az alkotásainkat. Ezért van a fában minden elemen, a hierarchia minden szintjén Refresh menüpont.)*

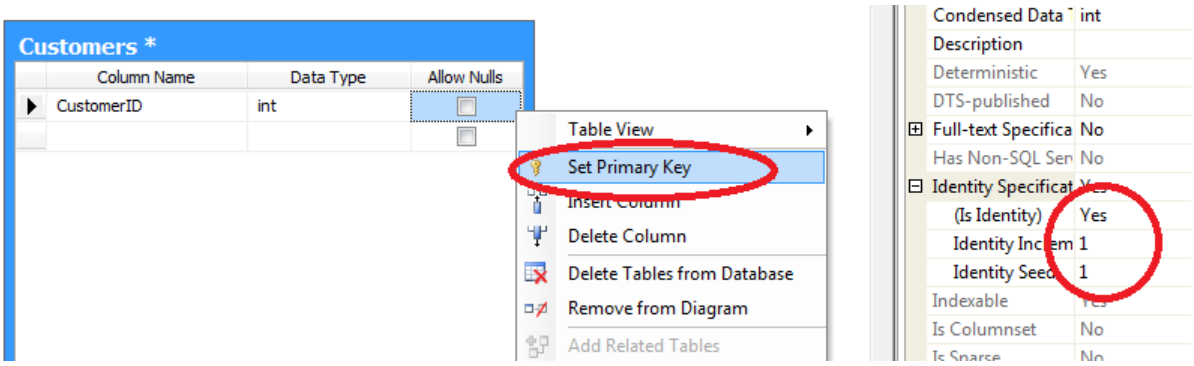
A Cities feladata egyébként az lesz, hogy amikor felveszünk ügyfeleket, akkor a városnév helyett a város egyedi azonosítóját tároljuk a Customers táblában, ezzel eleget téve egy csomó normalizálási szabálynak *(1, 2, és még a 3 is)*. Második táblánkat egy másik grafikus szerkesztőfelülettel, a Database Diagrammal hozzuk létre, hogy lássuk, hogy bizony kettő ilyenünk van. Keressük meg az adatbázis alatt a Database Diagrams ágat, nyissuk ki, a felbukkanó üzenetre természetesen olvasás nélkül mondjuk azt, hogy YES, és ezután már működik a jobbklikk a Database Diagrams ágon[3]. Az alábbi ábrára összemontíroztam a folyamatot, melynek során megjeleníti a létező tábláinkat *(mind az egyet, a Cities táblát)* és az Add gomb segítségével fel is vesszük azt a diagramra.

[3] *A kíváncsibbak kedvéért azt kérdezte, csinálhat-e táblát a diagramok elmentésére. Nyilván csinálhat, különben nem is működik ez a funkció.*



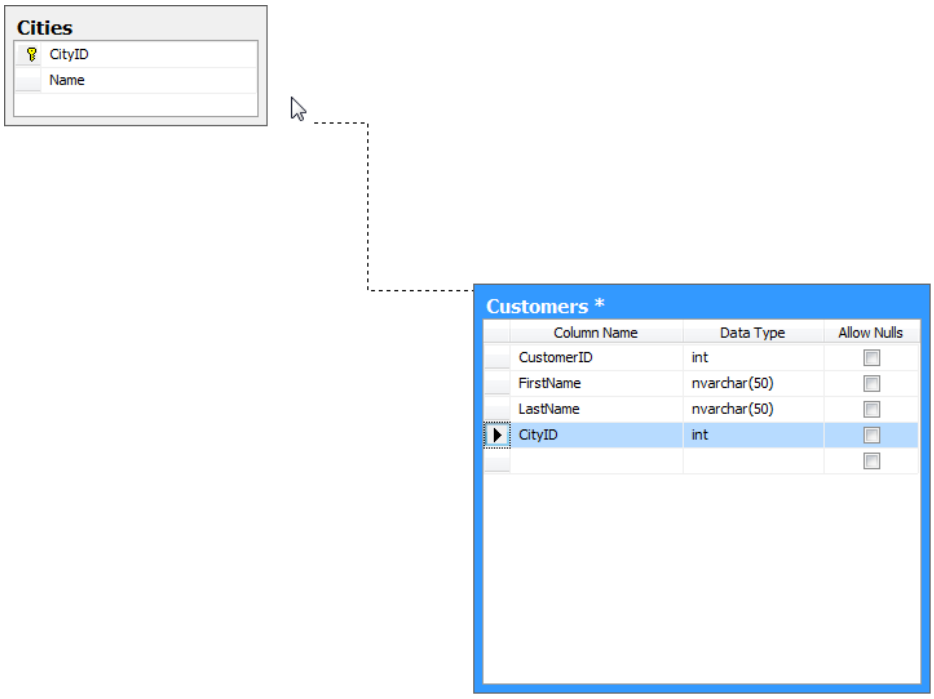
6. ábra - a diagramszerkesztő

Második táblánkat, a Customerst úgy hozzuk létre, hogy jobbklikkelünk a diagram egy üres részén *(konkrétan a semmin, az üres felületen)*, és a felbukkanó menüből a New Table… menüpontot választjuk. Ez a tábladesigner majdnem ugyanaz, mint az előző, de van pár különbség. Például ez előre bekéri az új tábla nevét (legyen Customers). A mezők tulajdonságát pedig ne alul keressük, hanem jobbra, de csak azután, hogy F4-gyel előcsalogattuk a Properties ablakot. Ismét egy összevágott jelenet következik, ahol épp a CustomerID mezőt csinálom meg IDENTITY-stül, PRIMARY KEY-estül:



7. ábra - kettes számú Table Designer

Ezt követően gyors egymásutánban készítsünk egy FirstName és egy LastName mezőt, mindkettőnek jó lesz az nvarchar(50) mezőtípus, és ne legyenek nullozhatók! Majd jön egy különlegesség. Megalkotjuk a vevők és a városok közötti kapcsolatot, kialakítva a két táblát összekötő referenciális integritási szabályt. Ennek az a menete, hogy készítünk egy CityID nevű, INT típusú, nem nullozható mezőt a Customers táblában, majd megfogjuk ennek a sornak a fejrészét, és rávontatjuk az egérrel a Cities táblára, így:



8. ábra - táblakapcsolat kialakítása

A Citiesen belül akárhol elejthetjük, mert úgyis felugrik egy ablak, hogy melyik mezőt melyikkel szeretnénk kapcsolatba hozni. Mivel furfangosan azonos mezőneveket használtunk *(ez a javasolt)*, nem sok dolgunk van a kiválasztással, csak az Enter ütlegelése.

Nyomjunk egy mentést *(floppy=mosogatógép)*, adjunk a diagramnak egy barátságos nevet, üssük az Entert, amíg visszabeszél, és ezzel nemcsak a diagramot mentjük el, hanem a táblát is legenerálja kapcsolatostul-mindenestül.

Akik mindeddig türelmetlenkedtek, hogy mikor jönnek már a scriptek, örömmel jelenthetem, hogy most. Ez a két tábla ahhoz kellett, hogy érdemben kipróbálhassuk az adatbázis sémájának scriptfájlba mentését, mert most már van valamink, sőt két valamink, amin ezt a funkciót be lehet mutatni.

1.4 Adatbázis scriptek

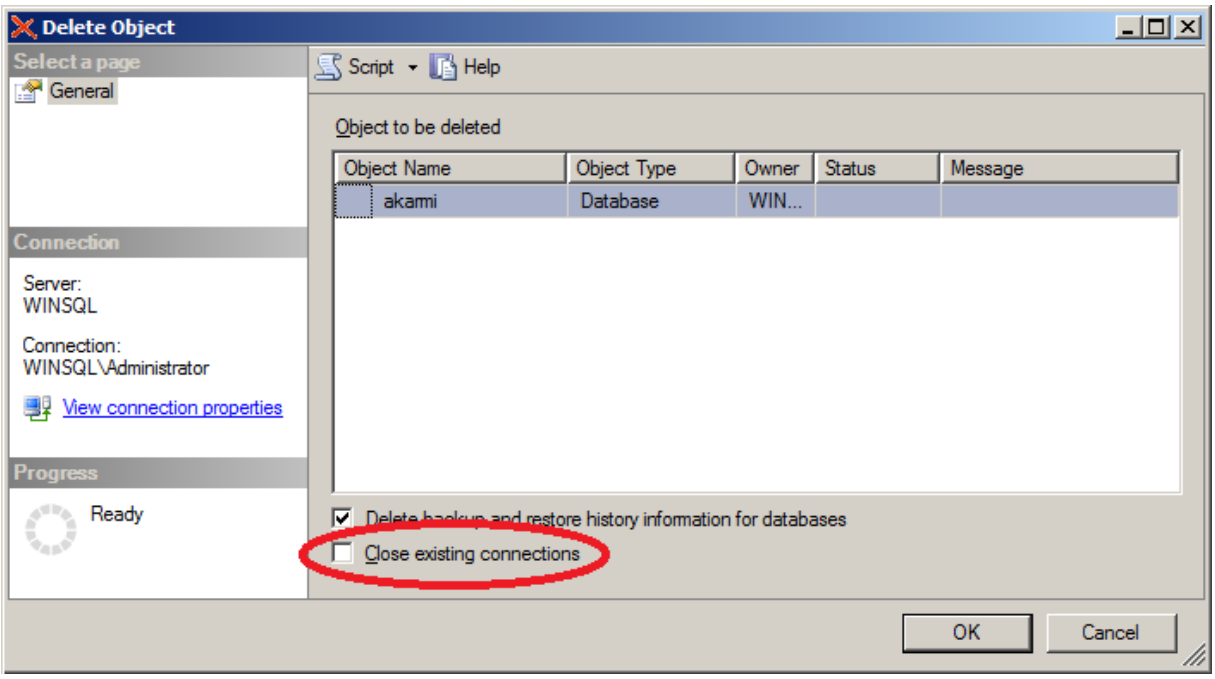
Jelenleg egy fejlesztői gépen dolgozunk. Valós környezetben egy plusz munkafázis szükséges ahhoz, hogy a kész adatbázist eljuttassuk a felhasználási helyére. Ehhez tipikusan scriptet használunk, mellyel 100%-os hűséggel le tudjuk képezni az adatbázis szerkezetét anélkül, hogy a fejlesztői gépen felvett nyamvadt tesztadatainkat is magunkkal cipelnénk, mint azt egy mentés-visszaállítás vagy egy lecsatolás-felcsatolás tenné. Vigyázat, Script menüpontból kettő is van, és az, amelyik odatolakodik a szemünk elé, nem az, amelyik ezt a funkciót tudja!

A Bank adatbázis lokális menüjében található egy tolakodó Script Database as… menüpont *(lásd a 2. ábrát)*, ami azonban csak és kizárólag a CREATE/ALTER/DROP DATABASE utasítást scriptelné le, a tábláinkat nem. Nekünk azonban a Tasks almenüben lévő Generate SQL Script menüpont kell, mert ez „mindent visz”. Indítsuk el!

A megjelenő varázsló első lapja szokásosan érdektelen. Next. A második lapon azonban már látszik, hogy ez egy mindenevő script lesz, noha egyelőre csak tábláink vannak. Ha lennének egyéb

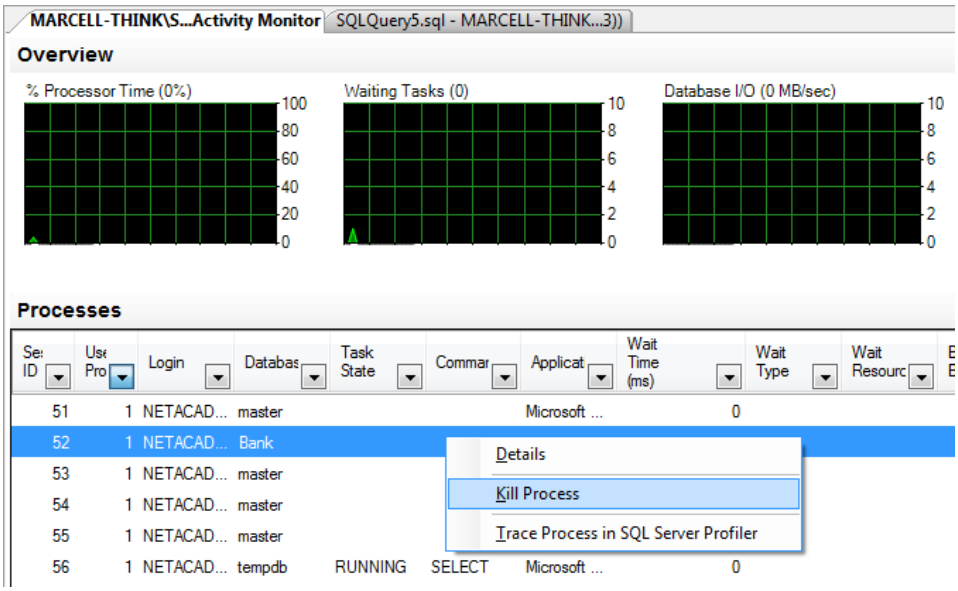
objektumaink, azokat is egyesével ki lehetne választani. Most maradjunk a minden kiválasztásánál, Next! A következő lapon válasszunk ki „célállomásnak" egy új lekérdezőablakot *(Save to new query window)*, Next, Next, Finish. Az eredmény egy csodálatos script, ami nemcsak azt a nulla darab beállítást tartalmazza az adatbázison, amit mi kiválasztottunk, hanem a default értékeket is, tehát tökéletes leírása jelenlegi adatbázisunknak. Táblástul. Aki nem hiszi, törölje bátran a bank adatbázist, majd futtassa le ezt a scriptet, és visszakapja ugyanazt az adatbázist.

Kitérő: Én is le szerettem volna törölni a magamét, de nem lehetett, mert a hibaüzenet szerint „a fiúk a bányában dolgoznak", tehát van egy rejtett kapcsolatunk a Bank adatbázis felé. Ha SMSS-szel törlünk, a párbeszédpanelen van alul egy kilövési opció:



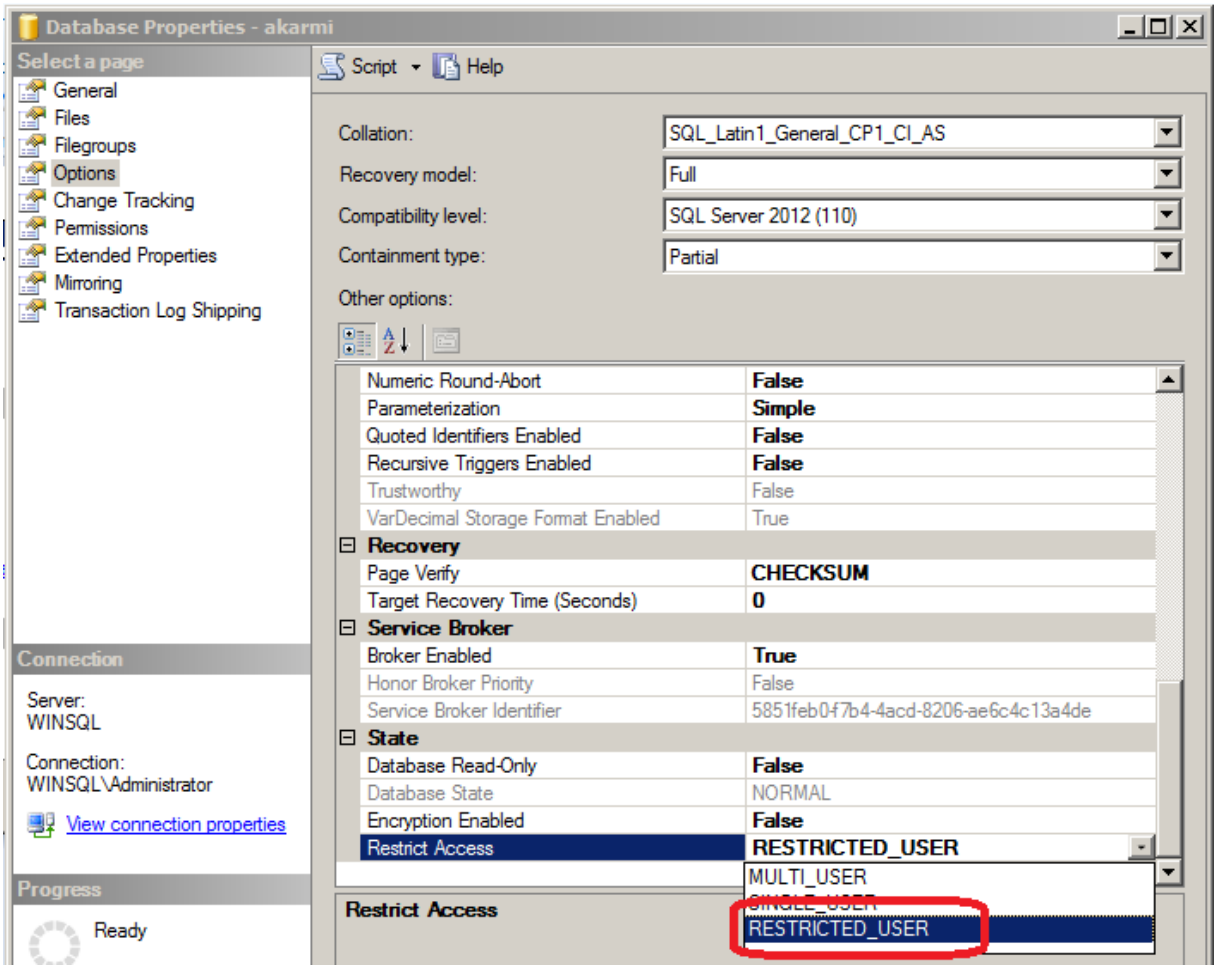
9. ábra - élő kapcsolatok leszaggatása adatbázistörlés előtt

Mivel a kapcsolatok megszakítására sok másik művelet esetén is szükség lehet, nézzük meg, hogyan tehetjük ezt meg az adatbázis letörlése nélkül! Ha minden lekérdezőablakot átállítunk az adatbázisról, és a fában is kisétálunk, becsukjuk, elvileg minden kapcsolatot elengedünk. Ha még mindig makacskodik, ki kell lőni azt a „valakit", aki potyázik az adatbázisban. Ehhez használjuk az SSMS ikonsoráról elindítható Activity Monitort, így:



10. ábra - folyamat kilövése

Persze ehhez hozzá kell tenni, hogy a megszaggatott kapcsolatok azonnal visszajönnek, ha csak nem állítjuk át az adatbázist egyfelhasználós üzemmódra, amit pont erre találtak ki!



11. ábra - Mari néni kiszorítása az adatbázisból

A script lefutása után visszakapott adatbázisból két dolog hiányzik: az adatok (szerencsére) és az adatbázis diagram (sajnos, merthogy az is adat).

1.5 Mit tud a Query ablak?

Ha már így belefutottunk a lekérdezőablakba, nézzük meg tüzetesen, mit tud. Nyissunk egy új ablakot a scriptünk mellé az eszközsoron található legnagyobb gomb, a New Query megnyomásával! *(Tetszőleges számú lekérdezőablakunk lehet egyszerre nyitva, amelyek mindegyike külön felhasználónak számít az SQL Server számára, így nem lát bele az összes többi kapcsolat tranzakcióiba. Hogy ez licencügyben mit jelent, ne firtassuk, mert fogalmam sincs! ☺)*

Ebbe az ablakba írjuk be életünk első (?) SQL-parancsát, egy SELECT-et! Jó egyszerű legyen, mindössze azokat a kulcsszavakat használjuk fel a SELECT parancsból, amiket kötelező – vagyis semmi mást, mint a SELECT-et! *(Általános tévhit, hogy a SELECT-tel mindenképpen táblából kell lekérdeznünk. Ez egyáltalán nincs így. Kérdezhetünk konstansokból, kifejezésekből, táblaértékű függvényekből stb.)*

```
SELECT 1
```

Mit tehetünk ezzel a „scripttel”?

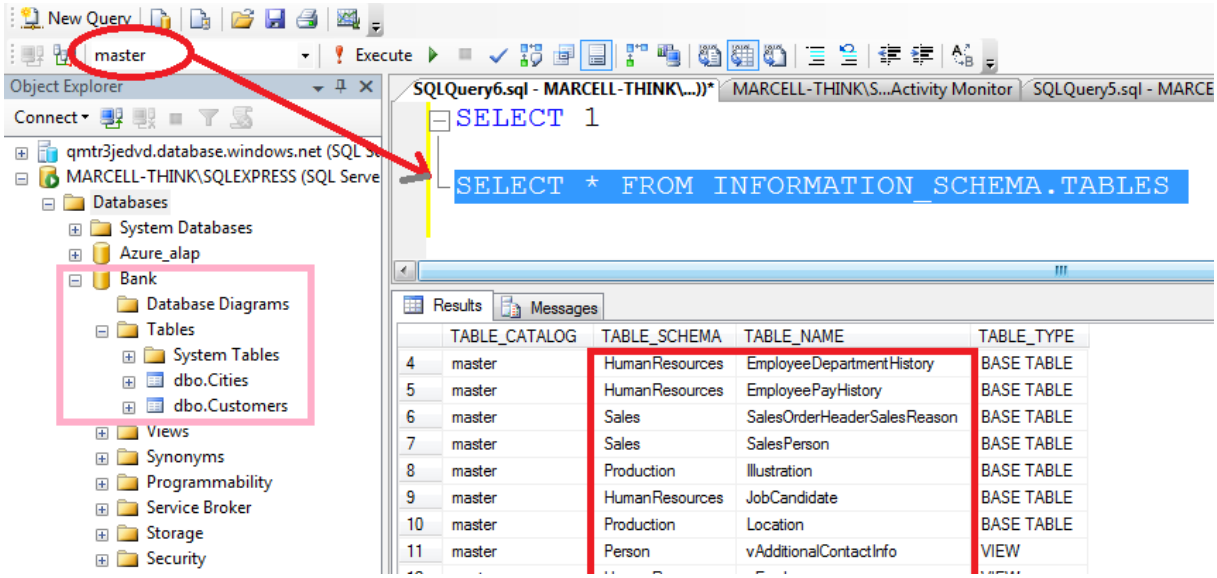
1. Lefuttathatjuk. *(Execute gomb az eszközsoron vagy F5, vagy a nagyon régi motorosoknak CTRL+E)* Ez a lekérdezés egy egysoros, egyoszlopos táblát ad vissza, melynél a mező és a tábla neve egyaránt üres.
2. Lépésenként futtathatjuk. *(Debug gomb vagy ALT-F5)* Ennek majd tárolt eljárásoknál, függvényeknél és triggereknél lesz jelentősége.
3. Ha túl sokáig tart a futása, leállíthatjuk a most épp szürke, de futás közben piros STOP gombbal.
4. Futtatás nélkül leellenőrizhetjük, hogy helyes-e a szintaxisa. *(Kék pipa gomb)*
5. Futtatás nélkül megnézhetjük, hogy ha lefuttatnánk, az SQL Server milyen végrehajtási terv mentén produkálná a sorokat. *(Felismerhetetlen ikon a kék pipától jobbra vagy CTRL+L, Estimated Execution Plan)*

Ez utóbbit gyakrabban használjuk, mint az ember gondolná. Egy igazi SQL-guru állandóan kíváncsi, mi zajlik a boszorkánykonyhában, mert általános esetben mi csak a parancsot adjuk ki, de hogy az SQL Server hogyan állítja elő boszorkányos ügyességgel az eredményhalmazt, az nem rajtunk múlik.

Hogy egy értelmes példát is nézzünk rögtön a végrehajtási tervre, kérdezzük le a tábláinkról tárolt információt az egyébként ANSI-szabványos metaadat-lekérdezési lehetőséggel!

```
SELECT * FROM INFORMATION_SCHEMA.TABLES
```

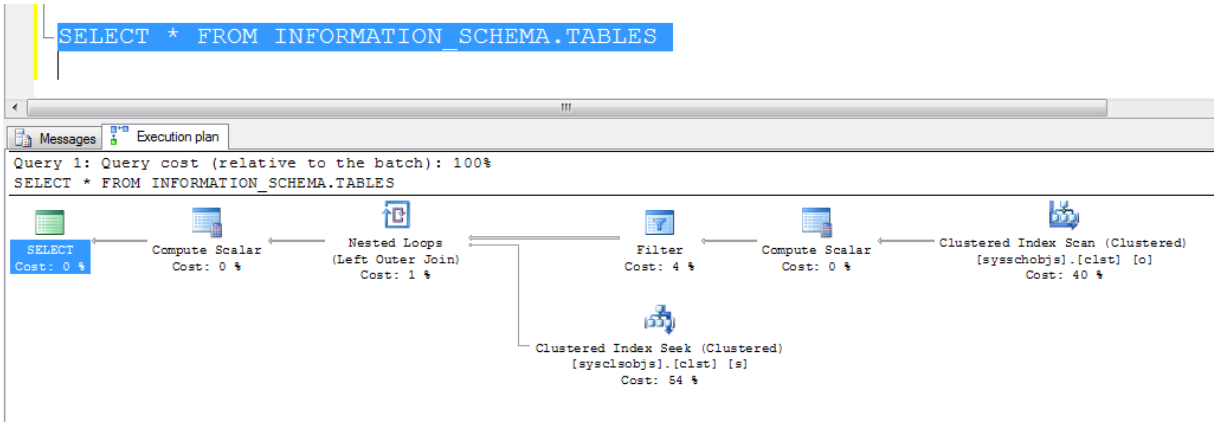
(Kitérő: Ha esetleg azt az eredményt kapjuk, hogy nincsenek tábláink, gyanakodjunk arra, hogy nem jó adatbázisban futtatjuk a lekérdezést! Tipikus hiba, és én mindig elkövetem, hogy – mivel alapértelmezésben a Master adatbázishoz kapcsolódunk – a Masterben keresünk dolgokat, vagy rosszabb esetben oda telepítünk táblákat, nézeteket. Az alábbi ábrán egy extrém példát láthatunk. A lekérdezés ugyanis azt mutatja, hogy „valaki” véletlenül a Master adatbázisba telepítette bele anno az AdventureWorks bicikliárusító példaadatbázist. ☺, Vajon hány éles rendszer fut így, mint ez itt? ☺)



12. ábra - adatok nem a legjobb helyen, a Masterben

Jegyezzük meg, hogy a „fa" fölött található adatbáziskiválasztó NEM a fára van hatással, hanem a lekérdezőablakra! Mindig! És egyben kivétel nélkül! *(Hogy akkor miért nem a lekérdezés felett van? – erről az ergó mókusokat kellene megkérdezni.)*

Ennek az ártalmatlan lekérdezésnek pedig ilyen a végrehajtási terve:



13. ábra - egy végrehajtási terv

Ne menjünk most bele részleteiben, hogy itt mi micsoda, csak csodálkozzunk rá a szépségére.

1.6 SQL-nevezéktan

Mielőtt elkezdünk eszeveszett tempóban további objektumokat létrehozni, röviden emlékezzünk meg az SQL-nevezéktannak hívott tudományról! Ez a tudományterület alapvetően azzal a kérdéssel foglalkozik, annak határait feszegeti, hogy milyen neveket adhatunk az SQL-objektumoknak. A tudomány mai állása szerint bármilyen butaságot kitalálhatunk, hívhatunk egy táblát, vagy akár egy mezőt „árvíztűrő tükörfúrógép"-nek, pusztán annyi a feladatunk, hogy az ilyen idétlen neveket [ ] zárójelek közé tegyük.

A nevezéktan másik nagy területe a foglalt szavak kérdése. Sajnos bármennyire is tetszik, hogy egy dátummezőt úgy hívunk, hogy Date, a tudomány álláspontja szerint mint név, ez is ostobaság, mivel az SQL-nyelvben a Date foglalt szó, egy adattípus, meglepő módon a dátum típus neve. És mit csinálunk a butaságokkal? Szögletes zárójelbe tesszük, és vidáman használjuk.

De vigyázat! Ez a tudomány nem foglalkozik azzal, miként áll fejre a kliensalkalmazás vagy a weboldal ilyen nevektől, úgyhogy vigyázzunk, mit fogadunk el a tanítások közül! Én például [Date] nevű oszlopot nem csinálnék, de mindenki szabadon azt tesz, amit akar.

Egy másik nagy filozófiai kérdéskör, hogy hogyan találunk meg egy objektumot az SQL-világban. Ha azt mondom, SELECT * FROM CUSTOMERS, ugyan milyen alapon merészeli az SQL Server megtalálni ezt a táblát?

Erre a kérdésre csak akkor tudunk érdemben válaszolni, ha megtanuljuk, hogy minden SQL-objektumnak valójában négytagú a neve, melyek közül hármat el tudunk hagyni, ha a környezeti feltételek ezt lehetővé teszik. A Customers tábla teljes neve valójában

kiszolgálónév.adatbázisnév.sémanév.objektumnév, azaz

MARCELL-THINK\SQLEXPRESS.Bank.dbo.Customers

melyből a szervernév elhagyható, mert épp erre a szerverre csatlakoztam *(SQL Servereken átívelő lekérdezések esetén ez nincs így!)*, az adatbázisnév elhagyható, mert ebben az adatbázisban állok épp, és a séma is elhagyható, mert itt részletesen nem taglalt okokból az objektumoknak általában dbo a sémája *(kivéve a kivételeket)*, így az egyetlen névdarabka, ami tényleg szükséges, az a hivatkozott objektum, jelen esetben a tábla neve.

1.7 További hasznos bigyóságok

Mielőtt végérvényesen elmerülnénk a scripttengerben, emlékezzünk még meg felsorolásszerűen azokról a további eszközökről, amelyek egy SQL-admin életét kényelmessé teszik, és hébe-hóba a fejlesztők is jó hasznát veszik!

- SQL Profiler: az SQL Serverre beérkező parancsokat kapja el, és teszi értelmezhetővé, elemezhetővé számunkra. Lassú lekérdezések kiszűrésétől a deadlockok elemzésén át indexek tuningolásáig sok mindenre használjuk.
- Database Engine Tuning Advisor: a Profiler jó barátja. A Profiler által összegyűjtött terhelésminta alapján javaslatot tud tenni egy optimális indexturmix kialakítására.
- SQLCMD: ez a parancssori eszköz mindent tud, amit az SSMS, csak az a kérdés, hogy aki használja, az tudja-e a szükséges parancsokat.
- BCP: a jó öreg parancssori kipi-kopi eszköz. A könyv írásának pillanatában ez a legintelligensebb eszköz helyi SQL-adatok felhőbe mozgatására, mert csak ez tudja az IDENTITY mezőket helyesen feltölteni (-E kapcsoló).

2 Data Definition Language

A nyelv elemei közül elsőként az egyes objektumok létrehozásáért felelős DDL-nyelvrészt tekintjük át. Ezt megelőzően picit elmerengünk az adatbázistervezés elméletén, az úgynevezett normál formákon, valamint a felhasználható adattípusokon.

2.1 Adatbázistervezés – dióhéjban (a tudományos alaposság igénye nélkül)

Amikor létrehozunk egy adatbázist, szinte kivétel nélkül a valóság egy darabkáját öntjük bitekbe. A mi célunk, hogy ez a lehető legellentmondásmentesebben történjen meg. Esküdt ellenségünk a redundancia, mert ha bármit két példányban tárolunk, mindig fennáll a veszélye annak, hogy csak az egyik példányt módosítjuk, ezáltal szétzilálva az adatok önellentmondás-mentességét, mert utána ki fogja megmondani, melyik adat az érvényes?

Ahogy a mondás tartja: addig, és csak addig tudod, hogy mennyi a pontos idő, amíg egyetlen órád van. Amint van kettő, többé el nem döntöd, mennyi az idő!

Redundanciából sokféle létezik. Vannak teljesen látható, és vannak rejtőzködő példányok. Mindkét típust irtjuk. A normalizálási szabályok lényege, hogy egyszerű módszerek betartásával kiirtsuk az ellentmondás lehetőségét az adataink közül. Normalizálási szabályból van vagy hat, melyek egyre kisebb redundanciákra lőnek. A gyakorlatban elegendő, ha az első hármat betartjuk, mert akkor már a tömegkatasztrófa biztosan elkerül minket. A normál formák szabatos leírása sok helyen olvasható, én most itt inkább a magyarról magyarra fordított, tehát érthető változatot teszem közzé, és még a sorszámokat is módosítom kissé, így lesz a háromból az alábbi négy:

1. Az adatbázis nem egy Excel tábla. Ne tégy fel egy lapra mindent! Válaszd külön az objektumaidat, és pakold őket külön táblákba! Ha kell, több tucat vagy akár több száz táblába. Ettől nem kell félni. A tábláidban szereplő egyedeket azonosítsd egy kulcsmezővel, melynél jó ötlet, ha a kulcs önmagában nem jelent semmit *(nem egy személyi szám vagy valami)*, mert ha nem jelent semmit, akkor később nemigen lesz olyan kényszer, hogy megváltoztasd, ja és mellesleg az értelmetlen kulcsok sokkal kisebbek tudnak lenni *(pl. egész számok)*, amelyekkel sokkal könnyebb lesz a táblákat összekapcsolni, és még az indexeket sem hizlalják feleslegesen[4].
2. A táblákban tárolt egyedek között kulcsmezők alapján teremts kapcsolatot! Mivel kapcsolatonként egyetlen értékpárod van, ebből következően nem az apa mutat az öt gyerekére az öt kezével, hanem mindig a gyermekek mutatnak egy kezükkel az apjukra. A gyermek kezét nevezd idegen kulcsnak, a kapcsolatot pedig referenciális integritásnak!
3. Ne tárolj olyan adatot a rekordokban, amelyek nem a kulcsmezőtől függnek! Rossz ötlet például a Customers táblában tárolni a városnevet és az irányítószámot, mert ebben az esetben az irányítószám a várostól függ és vice versa, semmi közük a kulcshoz[5].
4. Ne használj tovább bontható összetett mezőket. Ha van egy név meződ, bontsd fel vezetéknévre és keresztnévre, mert különben sohasem fogod tudni megmondani, hogy Sándor Béla Imrének melyik *(kettő?)* a keresztneve. *(Másrészről ez is problémákat vet fel. Ha nem lennének soknevű emberek, ez egy jó tipp lenne. De vannak. Prof. Dr. Phd. ifj. Kő Pál.)*

Miután ezt a sok okosságot mind bemagoltuk, továbbléphetünk az adattípusok felé.

2.2 Hétköznapi adattípusok

Két részre bontom az adattípusok taglalását. Ebben a részben a gyakran használt adattípusokat vesszük sorra, míg egy külön fejezetben a különleges adattípusokkal foglalkozunk.

2.2.1 Egész számok

Egész számból négyféle van az SQL Serverben:

- TINYINT (8 bit, 0..255 (nem előjeles))
- SMALLINT (16 bit, -32768..32767)
- INT (32 bit, mínusz kétmilliárd..plusz kétmilliárd)
- BIGINT(64 bit mínusz csillió..plusz kvadrillió)

Ezek közül manapság már talán csak a két nagyot használjuk. Tipikus felhasználási területük például az azonosítóképzés, amihez segítséget ad az IDENTITY() nevű függvény, mellyel kombinálva egy INT csodálatosan növekvő egyedi számlálót képez.

2.2.2 Pontos számok

Köztes lépés a lebegőpontos *(vagy lebegőpontatlan? ☺)* számok felé a tört számok tárolására is képes, de pontos adattípus, a DECIMAL, melynek ANSI-neve NUMERIC, és amelyikből két másik,

[4] Merthogy az elsődleges kulcs minden egyes indexbe bekerül, mert az indexek erre a kulcsra mutatnak. nem mindegy, hogy 4 bájtot ismételgetek a 14 indexemben, vagy 88 bájtot.

[5] Itt jegyzem meg, hogy adatmegtartási okokból el szoktuk követni ezt a „hibát”, mert nem mindegy, hogy egy áfás számla rekordba belemásolom a várost, és emiatt az ott soha többé nem változik meg, hiába variálnak a Város táblán, vagy hirtelen visszamenőleg megváltozik. A redundanciának is lehet gyakorlati haszna!

pénzügyi típus is származik, a MONEY és a SMALLMONEY. Ezekben az a közös, hogy ha – mondjuk – négy tizedesjegyig pontosak, akkor azt az életük árán is tartják, és ha valamilyen matematikai művelettel kirepülnénk ebből, aritmetikai hibával elszállnak. Ez kell ahhoz, hogy kerekítési hibák nélkül megússzuk a pénzügyeket. Egy DECIMAL mező a teljes számsor és a tizedesjegyek darabszámával adható meg, például így: DECIMAL(9,2). Ez egy maximum 9 jegyű számot takar, melynek maximum két tizedesjegye lehet – de ha valóban van két tizedesjegye, már csak 7 számjegye lehet az egészrészben.

2.2.3 Lebegőpontos számok

A FLOAT és a REAL típusok valók igazi matematikai műveletek végzésére. A FLOAT a nagy testvér, a REAL viszont feleannyi bájtot használ. A számtartományok mindkettőnél irdatlanok, lásd Books Online. Belső tárolásuk alap plusz hatványkitevő stílusú, kettes számrendszerben ábrázolódnak, ebből következően pont a tizedestörtekkel bajban vannak, mert a 0.1 csak végtelen kettedestört alakban tárolható kettes számrendszerben[6].

Az alábbi példa jól mutatja, mit nem tud a FLOAT, mert bizony 10 x 0.1 az csak 0.99999999999999999999999999999999999999999:

```
DECLARE @I FLOAT=0
WHILE @I<>1
BEGIN
      SET @I=@I+.1
      PRINT 'Végtelen ciklus? :-O'
END
```

Ebben az a fura, hogy ha kiíratom a ciklusban a@I értékét, akkor ott azt látom, hogy 1. Ezek szerint a kiíráshoz kerekít – de belsőleg biztosan nem. Mondjuk ki, hagyományos alkalmazások esetén a kettes számrendszerben tárolt lebegőpontos számok életveszélyt jelentenek.

(Blogger komment: ki az szerencsétlen, aki így hasonlít össze lebegőpontos számokat? Hát én, a példa kedvéért! ☺ A helyes megoldás az lenne, hogy – tudván a kerekítési mizériát – addig megyek a ciklussal, amíg a változóm és a célérték különbsége egy bizonyos határ alá esik, például WHILE 1-@I < 0.0001)

2.2.4 Szöveges típusok

Szöveges típusból van egy pár. Megkülönböztethetjük őket a tárolás módja szerint *(fix vagy változó hosszúságú)* vagy a karakterkészletük szerint *(hagyományos vagy unicode)*. De akárhogy is, mindegyik változat őstípusa a CHAR(x), mely x bájton tárolja a karaktereket, ha esik, ha fú. Ha nem teszünk bele x darab karaktert, kiegészíti szóközökkel, vele nem lehet kibabrálni. Ő viszont ezáltal kibabrál velünk, mert beleteszünk a mezőbe egy értéket, és egy másikat, a szóközökkel feltöltöttet kapjuk vissza. Az alábbi példát éppúgy érdemes bepötyögni és kipróbálni, mint az előző ciklusosat. Érdekes!

```
declare @a char(8)
set @a='a'
select @a, DATALENGTH(@a), @a+'itt a vége'
```

[6] Akit érdekel a „miért”, hát azért, mert a tízes számrendszer prímtényezői a 2 és az 5, ezek közül az ½ simán leírható kettes számrendszerben, ám az 1/5 csak végtelen kettedestörttel, a kettő szorzata, a 0.1 szintén nem, tehát pont a szokásos tizedek, századok nem.

```
select CHARINDEX(' ', @a)
```

Emiatt a viselkedés miatt a fix hosszúságú szöveges típusokat nem szeretjük, így mindjárt lecsökkent a szöveges típusok száma kettőre:

- VARCHAR(x) - sima, egybájtos típus, olyan szövegek tárolására kiváló, ahol ékezetek nemigen, de legalábbis nem soknyelvűen fordulnak elő. Például egy logfile.
- NVARCHAR(x) – unicode alapú, kétbájtos adattárolás, a világ összes nyelve belefér.

Most nézzük meg, mennyi lehet az x értéke ezeknél az adattípusoknál! Ehhez érintőlegesen meg kell ismerkednünk az SQL Server adattárolásával.

2.2.4.1 Lapok

Az SQL Server minden adatot úgynevezett lapokon (page) tárol, ami egyben az adatmozgatás, mentés alapegysége is. Minden lap mérete 8 kilobájt, se több, se kevesebb. Mivel általában minden adat a rekordok belsejében tárolódik, ez azt is jelenti, hogy egy rekord nem lehet nagyobb 8 kilobájtnál, sőt, egyetlen mező sem lehet nagyobb ennél, kivéve, ha ki van véve - a rekordból.

Tehát a VARCHAR(x) típusnál x 1 és 8000, NVARCHAR(x) esetén 1 és 4000 közé kell hogy essen ahhoz, hogy beférjen a rekordba. Ha ennél nagyobb lenne az adat, használhatjuk a

- VARCHAR(MAX)
- NVARCHAR(MAX)

típusokat, melyek ha túlnyúlnak, kiköltöznek a 8k-s lapról. Egyetlen valódi hátránya a (MAX)-nak, hogy az ilyen mezőket nem lehet indexelni – mivel kiköltöztek a rekordból, a lapról, mindenből.

2.2.5 Dátum- és időtípusok

Az SQL Server Gergely-naptár támogatása fenomenális. Itt és most mindenkinek megtiltom, hogy saját maga próbálja kiszámolni a jövő évet, a nyolc munkanapot vagy bármi más dátumszerűséget, mert az elszállás garantált, csak ki kell várni![7] Erre megvannak a beépített dátumtípusok és dátumfüggvények!

A dátum tárolása nagyon sokat fejlődött az SQL Server néhány korábbi verziójában. A kezdeti egyszerű DATETIME helyett jött a DATETIME2 és a DATETIMEOFFSET, melyek egyre nagyobb pontosságot, illetve ez utóbbi révén időzónakezelést is megvalósítottak. Ma, amikor elérhető közelségbe kerültek a felhők, és bármikor úgy alakulhat, hogy egy adatbázisunk néhány időzónával odébbköltözik, szerintem nem érdemes más típussal foglalkozni, mint a DATETIMEOFFSET-tel. A többi a múlt. Látni fogjuk később, hogy az időzónarész nem sok vizet zavar, viszont korrektté teszi a dátumokkal való játszadozást.

[7] Jé, hogy én hogy látom a jövőt! Amikor ezeket a sorokat írtam, 2012. február 26-a volt, épp utaztam Redmondba a szokásos éves MVP Summitra. A repülőgépen kalapáltam össze ezt a fejezetet. És három napra rá, február 29-én a Windows Azure mindenestől leállt, mert a szökőévet rosszul kezelte az egyik algoritmusuk. Kézzel dátumoztak. Ugye?

2.2.6 Bináris típusok

Bináris adatmezőben tároljuk a dolgozók fényképét, a PDF-dokumentumokat, a kottákat és a ZIP-fájlokat. A Microsoft Research készített egy tanulmányt[8], melyben kimutatják, hogy nem is olyan buta dolog fájlokat tárolni az SQL adatbázisokban. 1 MB-nál kisebb fájlok esetén az SQL Server kenterbe veri az NTFS-t. A dolgozók fotói tipikusan ilyen, picike képfájlok. SQL Serverbe velük!

Itt két adattípus közül választhatunk, BINARY és VARBINARY, erős preferencia a var-os változaton, és ennek a mérete is lehet (MAX).

2.2.7 Logikai típus

Logikai típusként a BIT jutott nekünk, ez van, de ez is elég. A BIT egy biten tárolódik, persze ha egy rekordban csak egyetlen BIT típusú mezőnk van, akkor nem, mert egyedi bitek tárolását sem a memória, sem a merevlemez nem tudja. Tehát ilyenkor egy bájtot foglal. De ha lenne esetleg még egy bitünk, az is elférne ezen a bájton, és még további hat.

Hány értéke lehet egy bitnek? Ugye, hogy kettő? Nulla vagy egy, igaz vagy hamis, férfi vagy nő. Kivéve az SQL Server bitjeit, mert azok háromállapotúak, ugyanis ez az adattípus is lehetővé teszi üres érték, NULL tárolását!

2.2.8 Globally Unique Identifier

Ha valaki olyan online bankot készít, mint amilyen a miénk is lesz, hamar szembesülni fog azzal a kényelmetlen dologgal, hogy a weblapon megjelenített rekordokat valahogyan azonosítani kell, de ezzel ügyelni kell, mert a weblapba ágyazott azonosítókat bárki megszerezheti, ha másképp nem, az oldal forráskódjából.

Ha a weblapon a csodálatosan automatikusan növekvő INT típusú egyedi kulcsot használjuk azonosítóként, akkor sajnos azt kockáztatjuk, hogy idióta hekkerek megpróbálnak más rekordokhoz hozzáférni, mint amihez joguk van, mert simán kitalálják más felhasználók azonosítóit (hisz egyesével növekvő számokról van szó), és úgy járnak, mint 2011-ben a Citibank. Már vagy 200-300 ezer bankszámla összes adatát olvasták el ügyes hekkerek az URL átirkálásával, mire a Citi észlelte (?) a problémát, és beavatkozott[9].

Kell lennie egy másik útnak, egy kitalálhatatlan, egyedi azonosítónak. Van is! Az adattípust úgy hívják, UNIQUEIDENTIFIER, és GUID típusú értékeket tárolhatunk benne, amit adott esetben elsődleges kulccsá is tehetünk. Új és új GUID-okat a NEWID() függvénnyel kérhetünk magunknak.

2.3 Játék a betűkkel, sorba rendezések

Mai adásunkban azt kérdezzük az itt megjelent játékosoktól, hogy vajon az egér azonos-e az égerfával vagy Eger városával. A válaszok pártatlan elbírálására, kedves nézőink, egy SQL Servert hívunk segítségül. Az első kérdés tehát így hangzik: az egér egyenlő-e égerrel? Óra indul!

```
SELECT 'igaz' WHERE 'egér'='éger'
```

[8] To Blob Or Not To Blob: http://research.microsoft.com/apps/pubs/default.aspx?id=64525
[9] http://www.theinquirer.net/inquirer/news/2079431/citibank-hacked-altering-urls

Alapbeállítások szerint az egér nem éger, megnyugodhatunk. *(Ha az egyenlőség igaz, a fenti lekérdezés kiírja, hogy 'igaz'. Ellenkező esetben üres halmazt kapunk.)* És mi a helyzet Eger és eGeR esetén?

```
SELECT 'igaz' WHERE 'Eger'='eGeR'
```

Úgy találjuk, ez az állítás igaz. Ez is megfelel az előzetes várakozásainknak. Csakhogy ez csupán egy alapértelmezett működés. Mind az ékezet, mind a kis- és nagybetű érzékenységet akár adatbázis, akár az egyes lekérdezések szintjén manipulálhatjuk.

Mit szólunk ehhez, itt lent?

```
SELECT 'igaz' WHERE 'egér'='éger'
COLLATE Hungarian_CI_AI
```

Hoppá! Ebben az esetben az egér bizony egy éger!

És ez hogy tetszik?

```
SELECT 'igaz' WHERE 'Eger'='eGeR'
COLLATE Hungarian_CS_AS
```

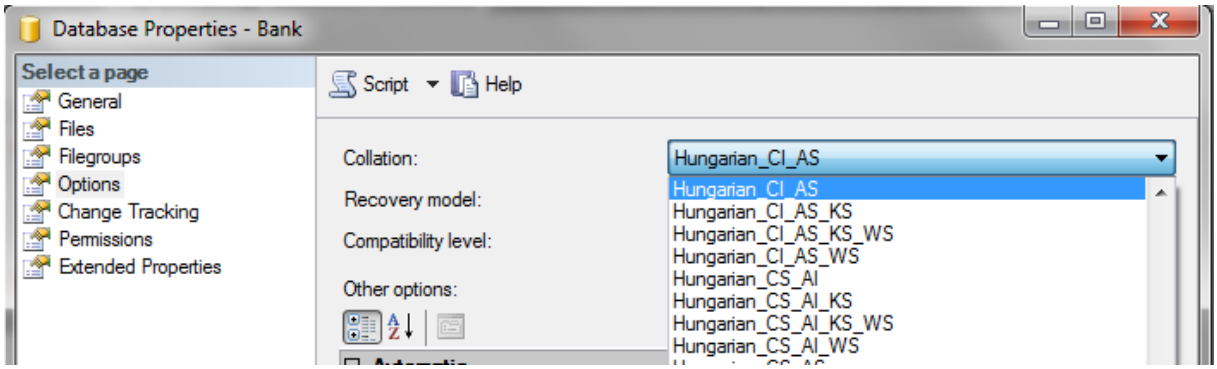
Ó! Nem egyenlők. Pedig csak egy icipicit gépeltük el a város nevét.

A legszebb, amikor valakinek majdnem minden lekérdezése helyes választ ad, kivéve a hosszú ő és ű betűket tartalmazó kereséseket, mert az meg nem. Ennek oka a Latin1_General karakterkészlet, ami jó sok európai nyelv jó sok ékezetes karaterét tartalmazza – kivéve a mi ő és ű betűnket. Ha ezekre keresünk, lecsapja az ékezetet, és tűzőr=tuzor. ☹

A változás a nyelvi beállításokban rejlik. Mindkét COLLATE utasítás végén szépen látszanak azok a módosítók, amelyek ezt a hatást kiváltották. A Hungarian jelentése: magyar *(tiriviális)*. Vajon mit jelent a CI és az AI, a CS és az AS? Case Insensitive, Accent Insensitive, Case Sensitive, Accent Sensitive. Ékezet- és kisnagybetű érzéketlen és érzékeny.

Amíg nem írjuk ki, addig az adatbázis alapértelmezett COLLATION-jét használja az adatbázismotor, ami telepítéskor alakult ki a Next-Next-Finish veszett nyomogatása közben, és alapból az operációs rendszer hasonló beállítását veszi át.

Ha valaki feltelepít egy angol nyelvű Windowsra egy SQL Servert, majd mégiscsak magyar sorba rendezést szeretne használni, átállíthatja a nyelvi alapbeállítást az adatbázison. Ezt jó előre érdemes megcsinálni, mert visszamenőleges hatása nincs, amit már létrehoztunk (táblák, mezők), azok változatlanul maradnak. Csak az új adat lesz magyar. A nyelvi beállítás itt változtatható meg az adatbázis tulajdonságlapján:



14. ábra - az adatbázis alapértelmezett nyelvi beállítása

Ezen felül a „játék a betűkkel" kategóriába tartozik a sok-sok stringfüggvény, melyekkel keresni, vágni, balról-jobbról lecsapni, egyszóval mindent lehet, amire úgy nagy vonalakban számít minden programozó lelkületű polgártársunk. Javaslom a helpben felütni a „String functiuons" fejezetet.

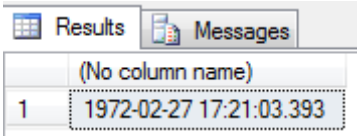
2.4 Hány éves vagyok? És hány hónapos, napos, perces, órás, másodperces?

A Gergely-naptár az, ami. Már egyszer korábban megtiltottam a saját készítésű dátumgörcsölést, amibe csakis belebukhat az ember. Viccesen hangzik, de ez is megtörtént eset: „Küldj virágot Szökőnap nevű ismerőseidnek!"

Ez ellen véd, ha nem mi magunk szöszmötölünk a dátumokkal, hanem rábízzuk Gergelyre, a naptárra. Első feladatként számolja ki mindenki, hány éves. Intuitív megközelítés (nem valós szülinappal, fele ilyen idős sem vagyok):

```
SELECT GETDATE()-'1940.01.01'
```

Mert ugye a matek az matek, a kivonásnak működnie kell. Végül is kapunk valamilyen eredményt, panaszra nem lehet okunk, bár hogy mi ez az eredmény, a jó ég sem tudja:



De hogy nem életkor, az bizonyos. Leszűrhetjük a következtetést, hogy dátumokat nem így kell egymásból kivonni. Hanem akkor hogyan? Hát dátumfüggvényekkel! Van ugyanis egy direkt dátumkivonásra kifejlesztett függvény az SQL Serverben, a DATEDIFF(), ami még azt is kezeli, hogy milyen mértékegységben kérjük a különbséget. Évben? Napban? Másodpercben? A fenti feladat helyes megoldása:

```
SELECT DATEDIFF(YEAR, '1940.01.01', GETDATE())
```

Házi feladat a helpben utánanézni a DATEPART függvénynek, ahol le van írva részletesen, hogy milyen egyéb mértékegységek léteznek a YEAR-on kívül.

Ezzel megvolnánk. Most számítsuk ki a mai naphoz képesti nyolcadik napot! Ebben az esetben merő véletlenségből jól működik az intuitív megoldás is *(SELECT GETDATE()+8)*, én azonban most is a

megfelelő függvényt *(DATEADD())* használnám, mert abban tisztán olvasható, hogy nyolc mit akarunk hozzáadni. Évet? Napot? Hónapot?

```
SELECT DATEADD(DAY, 8, GETDATE())
```

Rendben. És hogyan számolnánk ki a pontosan egy évvel ezelőtti napot? Vajon negatív számokat is kezel ez a függvény? Igen.

```
SELECT DATEADD(YEAR, -1, GETDATE())
```

Menjünk tovább. Mi lenne a következő hónap első napja? 2012-ig bezárólag ezt úgy kellett kiszámítani, hogy kivettük a dátumból a napok számát DATEPART() függvénnyel, azt kivontuk a mai napból, és hozzáadtunk egy napot, majd még egy hónapot. SQL 2012 esetén a helyzet sokkal egyszerűbb, mert van EOMONTH() függvény, ami megadja a hónap végét, amihez már csak egyet kell „aludni”, és megvan a következő hó eleje.

```
SELECT EOMONTH(GETDATE())+1
```

2.5 Az a mocsok NULL

Adattípus-körbejáró fejezetünkben nem mehetünk el szó nélkül az üres értékek, a NULL mellett, amely annyi kezdőnek keseríti meg az életét. Nem találja meg őket a keresésekben, nem lehet tudni róla, hogy valódi NULL vagy generált, és még sorolhatnánk. Ezek miatt a problémák miatt is javasolt a nullozható mezők számát a minimumra szorítani. Másrészt persze az is igaz, ha van egy kötelezően kitöltendő mező, amibe nincs mit írni, a felhasználók hülyeségeket fognak oda bevinni, kényszerűségből. Mert dolgozni kell.

Az az adat, ami csak néha van, az nem az objektum saját tulajdonsága, az egy gyermekrekord inkább. Altáblánál normális viselkedés, hogy egy gyermekrekord vagy van, vagy nincs.

De ha már vannak NULL-jaink, nézzük meg, hogyan bánjunk el velük! Alaphelyzetben, ha nem állítgatunk semmit, hanem maradunk az ANSI-szabványos alapbeállításnál[10], igaz lesz a következő szabály:

A NULL SEMMIVEL SEM EGYENLŐ, MÉG ÖNMAGÁVAL SEM!

Aki ezt megjegyzi, túl fogja élni a NULL-ok támadását.

Nézd meg az alábbi példát:

```
SELECT 'igaz' WHERE NULL=NULL
```

Nos, ez nem igaz! Üres halmazt kapunk!

A fentebb kiemelt főszabályból tehát az következik, hogy ha valaki olyan SQL-kódot lát, ahol a NULL egy egyenlőségjelnek támaszkodik, az a kód ROSSZ, nem is kell tovább elemezni, HIBÁS, VACAK.

Na jó, de akkor hogyan hasonlítunk össze értékeket NULL-lal? Csakis az erre rendszeresített IS NULL művelet segítségével! Az előbbi példa helyesen:

[10] A teljesség kedvéért jegyezzük meg, hogy ki lehet lépni az ANSI világából a SET ANSI_NULLS OFF paranccsal, és akkor NULL=NULL! A további leírásban arra támaszkodunk, hogy nem vagyunk mi ilyen fejlettek, nem kapcsolunk semmit sehova. Ami meg is felel a gyakorlatban követett cselekvési sornak.

```
SELECT 'igaz' WHERE NULL IS NULL
```
Ez már igaz!

Viszont ebből az is következik, hogy ha egy lekérdezésben két mező értékét hasonlítom össze, és azok történetesen üres mezők, hibás eredményt kapok. Röviden: igen. Hosszabban: ilyen esetekben cselezni kell, fel kell turbózni a WHERE-feltételt a nullos helyzetre, valahogy így:

```
SELECT * FROM A
WHERE A=B OR (A IS NULL AND B IS NULL)
```
Őszintén sajnálom.

Ha valaki sorokat számol, gondoljon arra, hogy a COUNT(Mező1) átugrálja a NULL-okat, míg a COUNT(*) beszámolja. És még sorolhatnánk. További NULL-os problémákkal a JOIN-os fejezetben találkozunk.

2.6 CREATE TABLE

Azt hiszem, eleget tudunk ahhoz, hogy el tudjuk készíteni a többi táblát, amire bankunknak szüksége van. Először is kell egy Accounts tábla, ami a Kedves Ügyfél bankszámláját reprezentálja. Aztán szükségünk lesz egy Transactions táblára, ami a bankszámlaműveleteket gyűjtögeti. Mi kell tehát egy bankszámlatáblába?

- Egyedi azonosító, ami nem a bankszámlaszám, hanem csak egy sima INT. Nem lehet NULL, de legyen szíves automatikusan növekedni, és kulcsmezőként viselkedni!
- Ügyfélkód, ami a Customers táblára mutat. Ez sem lehet üres, olyan nincs, hogy egy bankszámla senkihez sem tartozik.
- Bankszámlaszám, ami valójában egy szöveg, nem maradhat üresen, viszont tutira sohasem lesz benne ékezetes karakter.
- Egyenleg - az pont nem, mert azt a tranzakciók összesítéséből nyerjük. Az állandó, minden egyenleglekérdezésnél lefutó brutális kiszámítgatást később korrigáljuk.
- Létrehozás dátuma – ezt a mezőt automatikusan fogjuk töltögetni, naplózási célból, és okosan olyan dátumformátumot választunk, ami időzónahelyes.
- Egy krikszkraksz kód, ami a számlát úgy azonosítja, hogy bátran kitehessük az azonosítót a webre. Talán mondanom sem kell, hogy ez sem lehet üres. Ja, és a kutya sem akar a GUID-okkal egyesével vesződni, minden rekordban jelenjen meg magától egy új GUID!

Transact SQL-ül:

```
CREATE TABLE Accounts(
      AccontID INT PRIMARY KEY IDENTITY NOT NULL,
      CustomerID INT NOT NULL
            FOREIGN KEY
            REFERENCES Customers(CustomerID),
      AccountNumber VARCHAR(88) NOT NULL,
      CreationDate DATETIMEOFFSET NOT NULL
            DEFAULT GETUTCDATE(),
      URLCode UNIQUEIDENTIFIER NOT NULL
            DEFAULT NEWID()
```

```
)
```

Ha lenne már ügyfelünk, fel is vihetnénk egy bankszámlát. Ha lenne már városunk, fel is vihetnénk egy ügyfelet. A referenciális integritás szépségei ☺!

Új város:

```
INSERT Cities VALUES('Budapest')
```

Új vevő:

```
INSERT Customers VALUES('Jakab', 'Gipsz', 1)
```

Új bankszámla *(ez az eddigiekhez képest bonyolultabb, mert itt ki akarom hagyni a dátum és az URLCode mezőt, tehát fel kell pontosan sorolnom, hogy melyikeket töltöm ki)*:

```
INSERT Accounts(CustomerID, AccountNumber) VALUES(1, '12345678-
64353453')
```

És az eredmény:

Results | Messages

	AccountID	CustomerID	AccountNumber	CreationDate	URLCode
1	1	1	12345678-64353453	2012-02-26 18:24:50.8330000 +00:00	F76EF604-20F2-419D-999B-E3B036C3DFD8

Vagány!

Jöhet a Transactions tábla. Mi kell bele?

- Egyedi azonosító, nem lehet NULL, de legyen szíves automatikusan növekedni, és kulcsmezőként viselkedni!
- Számlaszám kapcsolatmező, ami az Accounts táblára mutat. Ez sem lehet üres, olyan nincs, hogy egy tranzakció semelyik bankszámlához sem tartozik.
- A tranzakció másik résztvevőjének neve. Akitől pénzt kaptunk vagy akinek adtunk. Nem lehet üres, de bőven lehet benne ékezetes karakter. Az Ománból kapott pénzt továbbutaljuk Hongkongba.
- Mennyiség, ami megmutatja, hogy mekkora összeg mozgott. Lehet negatív szám is, tört szám is, de az nem lenne baj, ha a 0.1 az nem 0.099999999999 lenne.
- Pénznem. Ebbe a mezőbe a pénznem kódját kell beírni, például HUF, EUR stb. Kötelező mező ez is. A pénznem mindig három karakteres, akármi történjék is.
- Létrehozás dátuma – ezt a mezőt automatikusan fogjuk töltögetni, naplózási célból, és okosan olyan dátumformátumot választunk, ami időzónahelyes.
- Egy krikszkraksz kód, ami a tranzakciót azonosítja, hogy bátran kitehessük az azonosítót a webre.

```
CREATE TABLE Transactions(
      TransactionID INT PRIMARY KEY IDENTITY(1,1) NOT NULL,
      AccountID INT NOT NULL FOREIGN KEY(AccountID) REFERENCES
      dbo.Accounts (AccountID),
      Partner NVARCHAR(88) NOT NULL,
      Amount MONEY NOT NULL,
      Currency CHAR(3) NOT NULL,
      CreationDate DATETIMEOFFSET(7) NOT NULL DEFAULT (getutcdate()),
```

```
    URLCode UNIQUEIDENTIFIER NOT NULL DEFAULT (newid())
)
```

Hogy a későbbiekben ne a semmit kérdezgessük nagy erővel, gyorsan futtassuk le az alábbi parancsokat, hogy létrejöjjön Kő Benő is, aki Gödöllőn lakik. Ha megfigyeljük a tranzakciókat, egy nemzetközi összeesküvés kellős közepén találhatjuk magunkat. Gipsz Jakab jelentős összeget kapott a Lehman Brotherstől, amely azonban Kő Benőtől származott! ☹

```
INSERT Cities VALUES('Gödöllõ')
INSERT Customers VALUES('Benõ', 'Kõ', 2)
INSERT Customers VALUES('Damon', 'Hill', 2)
INSERT Accounts(CustomerID, AccountNumber) VALUES(2, '9876543-
64353453')

INSERT Transactions(AccountID, Partner, Amount, Currency) VALUES
(1, 'Lehman Brothers', 100000000, 'USD'),
(1, 'Enron', 2370000000, 'EUR'),
(1, 'Postabank', 888888, 'HUF'),
(2, 'Lehman Brothers', -100000000, 'USD'),
(1, 'Fuggers International', 8768700000, 'USD'),
(1, 'Uncle Sam', 4, 'USD')
```

2.7 IntelliSense

Ha most jól megnézzük a bal oldali fában, miket alkottunk, nem látszanak. Mivel nem az SSMS-szel csináltuk a táblákat, nem is vette őket észre. Egy Refresh a Tables ágon majdnem mindent megold, azonban a lekérdezőablakban minden új objektumunk nevét pirossal aláhúzva látjuk. A másik szereplő, aki nem vette észre, hogy új objektumokat hoztunk létre, az az IntelliSense. Rajta pedig egy CTRL+SHIFT+R segít.

2.8 Számított mezők

Vannak olyan mezők, melyeknek nem mi adunk értéket, hanem az SQL Server. Láttunk már ilyet, az IDENTITY jó példa erre. Egy másik lehetőség, ha a tábla egyik sorában szerepeltetünk egy számított mezőt. Olyan értékekre gondolok, amiket nagyon gyakran kell lekérdezni, viszont nem önálló értékek, például mint az áfás ár. Tiszta redundancia, az igaz, de a redundáns érték karbantartása az SQL Server részéről biztosított. Az áfa talán nem a legjobb példa, mert félévenként változik, akkor legyen egy bankkártyajutalék, amit fixen 3%-nak veszünk.

Így készíthetünk egy számított mezőt a Transactions táblába:

```
ALTER TABLE Transactions ADD CardFee
    AS Amount * .03
```

Ez az utasítás létrehoz egy plusz mezőt a táblában, amely azonban nem tárolódik, hanem röptében kiszámítódik, valahányszor egy lekérdezés érinti. Házi feladat utánanézni, hogyan lehetne elérni, hogy

ez a számított mező tárolódjon is, vagyis még csak kiszámolnia se kelljen az SQL Servernek, hanem az adat azonnal rendelkezésre álljon a lekérdezések számára.

2.9 Átnevezések

Egy rövid kitérő erejéig emlékezzünk meg az átnevezésekről! Egy fejlesztés során számtalan esetben előfordulhat, hogy egy mezőt vagy egy táblát át kell nevezni. Erre mindenki mondhatja, hogy nem jó ötlet, mert az átnevezett tábla kiesik a rá hivatkozó tárolt eljárások és nézetek alól, és ez igaz is. Ettől még az átnevezés igénye fennáll, ebből nem engedek. Erre a feladatra az sp_rename tárolt eljárást használjuk, merthogy az ANSI SQL-ben nincs erre a célra semmiféle parancs. Egy tábla átnevezése így néz ki:

```
SP_RENAME 'RegiNev', 'UjNev'
```

Mezőt átnevezni pedig a következő okos módon lehet:

```
SP_RENAME 'Tablacska.Mezo', 'UjMezonev', 'COLUMN'
```

Szép kis szintaxis!

3 Data Query Language I. Egyszerű SELECT utasítások

Az SQL-nyelv legfontosabb része a lekérdezőnyelv. Minden mást előjátéknak tekinthetünk, amely csupán ahhoz kell, hogy legyen mit lekérdeznünk. Ennek a résznyelvnek az univerzális kulcsszava a SELECT, amelyre az SQL Server harap, és ami ez után következik, azt megpróbálja lekérdezésként értelmezni.

3.1 A SELECT utasítás. Mi az a csillag?

A legegyszerűbb SELECT-utasítás, amit mindenki ismer, még azok is, akik soha életükben nem láttak adatbáziskezelőt, a

```
SELECT * FROM Tábla
```

Ez az igen egyszerű változat nagyon elterjedt, mert ezt még úgynevezett programozók is ki tudják magukból szenvedni. Aztán így, ahogy van, jól beleírják a programjukba, majd csodálkoznak, hogy az alkalmazásuk az adatok szaporodásával egyre csak lassul.

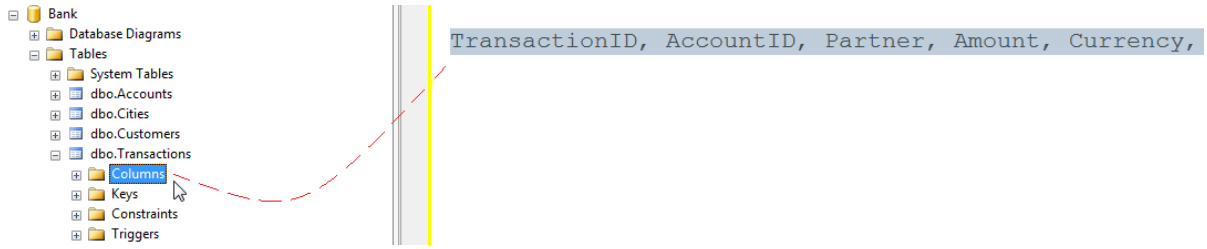
Vajon hány sort kérdez le ez az utasítás egy ötmillió soros táblából? Mindet!

Vajon hány oszlopot kérdez le egy olyan táblából, melyben mindenféle bináris oszlopok vannak fotókkal és PDF-ekkel? Mindet!

Én azt javaslom, hogy éles adatbázison lehetőleg soha ne használjuk ebben a nyers, egyszerű formában a lekérdezést, mert meglepetésszerű problémákat okozhatunk önmagunknak. Mondjuk a kiadott sok Shared Lock miatt nem lehet gyógyszert kiírni a haldokló betegnek és hasonlók.

A minimum, ami kell bele, az vagy egy lekérdezési feltétel a WHERE kulcsszó után, vagy a TOP kulcsszóval csak bizonyos számú sor lekérdezése *(ez utóbbi trükköt használja az SSMS, amikor jobbklikk, ide-az-adatot menüpontot választjuk)*. Nem meglepő, hogy a WHERE kulcsszóval külön alfejezetben foglalkozunk. Minden pénzt megér egy olyan lekérdezés, ami csak azt adja vissza, amire valóban szükségünk van.

A másik fontos dolog, hogy ne kérdezzük le azokat az oszlopokat, amelyekre valójában nincs szükségünk. A fő bűnös a csillag (*) karakter, ami pont azt teszi, hogy válogatás nélkül az összes mezőt ránk borítja. Úgy is szoktam fogalmazni, hogy a csillag tiltott karakter az SQL-nyelvben, melyet lustaságból lépten-nyomon használunk. De hogy is ne használnánk, mikor a megfelelő mezők álladó kiirkálása annyira fárasztó? Ennek a lustasági problémának az áthidalására hadd mutassak egy trükköt! Az SSMS olyat tud, hogy ha egy tábla alatti Columns mappát behúzol a lekérdezőablakba, beírja a kurzor helyére az összes mezőnevet, neked meg már csak törölnöd kell a nemkívánatosakat!



15. ábra - mezőnevek legenerálása egy egérmozdulattal

Még egy fontos dolgot kell tudnunk a csillagról: fedőindex-killer. Létezik egy olyan fogalom az SQL Serverben, hogy fedő index. Ez azt az esetet jelenti, amikor egy lekérdezés minden mezőjét „lefedi" egy index tartalma, tehát nem kell lemenni a táblához az adatokért, minden megtalálható a táblánál jóval kisebb és jóval kevésbé macerált indexben. Nos, a csillag eleve lehetetlenné teszi, hogy az SQL Server ezt a fantasztikus gyorsítótrükköt alkalmazza.

3.2 Mezőlista, kifejezések a mezők helyén, aliasok

A csillag helyén nemcsak mezőnevek állhatnak a felsorolásban, hanem kifejezések is. Egy kifejezés lehet egy sima konstans, egy matematikai kifejezés, egy függvénnyel megdolgozott adat vagy akár egy komplett *(beágyazott)* SELECT utasítás. Az alábbi példán az első „mező" egy konstans, a második egy függvény, a harmadik egy „matematikai" kifejezés, a negyedik egy beágyazott SELECT *(amit mindig zárójelek közé kell írni, erről ismeri fel a fordító a beágyazást)*:

```
SELECT 1, 'Gipsz'+' '+'Jakab', GETUTCDATE(), (SELECT 1)
```

Ha most ezt lefuttatjuk, a következő szépséges eredményt kapjuk:

	(No column name)	(No column name)	(No column name)	(No column name)
1	1	Gipsz Jakab	2012-03-04 20:30:32.483	1

16. ábra - négy mező, egyik sem táblából származik

Hopp, nem maradt le valami? Egyik mezőnknek sincs neve! Ezen könnyen segíthetünk a mező ALIAS-ok bevezetésével. Minden mezőt tetszőlegesen át lehet nevezni röptében, a táblákból érkezőket is. A lekérdezésünk szépítésének következő lépése a keresztelő. Minden mezőnek nevet adunk:

```
SELECT 1 AS Sorszam, 'Gipsz'+' '+'Jakab'
AS Nev, GETUTCDATE() AS Datum, (SELECT 1) AS Osszeg
```

És az eredmény:

	Sorszam	Nev	Datum	Osszeg
1	1	Gipsz Jakab	2012-03-04 20:33:18.417	1

A csillag egyébként maga is csak egy „mező" a mezőlistában. Sokan meglepődnek, ha a csillag után vesszővel felsorolva további dolgokat kérdezünk le, pedig ez tisztán lehetséges.

```
SELECT *, 'akarmi', getutcdate() FROM Tábla
```

Ezzel a módszerrel ki tudjuk egészíteni a táblánk eredményhalmazát további, nem létező mezőkkel. Hogy egy „gyakorlati" példát is hozzak erre, tételezzük fel, hogy ki akarunk sorsolni tíz Hummert a

bank ügyfelei között. Véletlenszerűen szeretnénk kiválasztani a nyerteseket. Erre *(sok egyéb más lehetséges megoldás mellett)* ez a módszer alkalmas, ahol a csillag mellé plusz mezőként magát Fortuna istenasszonyt idéztük meg:

```
SELECT TOP 10 NEWID() AS Fortuna, * FROM Customers
ORDER BY Fortuna
```

A NEWID() függvény minden egyes sornál más és más értéket ad az egyes sorok mellé, és ha ez alapján „sorba rendezzük" az eredményhalmazt, egy véletlenszerű sorrendhez jutunk. Ebből kiválasztva az első tíz ügyfelet (TOP 10), már meg is vannak a nyerteseink!

3.2.1 ROW_NUMBER és társaik

Van egy adag függvény, amelyek segítségével okos plusz oszlopokat tudunk generálni az eredményhalmazunk mellé, ezek közül a ROW_NUMBER olyan, amelyiknek azonnal kézzelfogható gyakorlati haszna van. Tegyük fel, hogy egy listában növekvő sorszámokat kell a rekordok elé biggyesztenünk! A ROW_NUMBER() pont erre való. Az alábbi kód ennek használatára mutat egy példát:

```
SELECT ROW_NUMBER() OVER(ORDER BY Lastname)
AS Sorszam,* FROM Customers
```

Első ránézésre rémisztő, és másodikra is az, és azért ilyen, mert egy titkos összeesküvés alapján helyezi el a sorszámokat. Meg kell nekik adni egy titkos, zárójeles, semmi másra nem ható sorba rendezést. Ez a végső eredményhalmaz sorrendjére nem, csak a kiosztott sorszámok kiszórására lesz hatással.

A ROW_NUMBER() függvény egy nagyobb család része. Ide tartozik még a RANK() *(dobogós helyezés kiosztása)*, DENSE_RANK() *(dobogó, holtversenyt figyelembe véve)* és az NTILE() *(fürdőszobafüggvény, dekorációs célból használjuk, megmondja, hogy melyik rekordot melyik „csempére" kell „felragasztani")*, melyeket házi feladatba adok, mert nem gondolnám, hogy különösebben hirtelen bárkinek is szüksége lenne rájuk.

3.3 A FROM. Miből lehet szelektálni? Kell-e egyáltalán?

Következő kulcsszavunk a FROM. Sok-sok oldallal korábban már láttuk, hogy a FROM nem kötelező kulcsszava a SELECT-utasításnak. Lekérdezhetünk a kibertérből is adatokat *(pl. konstansok, függvények, változók)*. Normális lekérdezések esetén azonban szinte biztos, hogy van egy FROM kulcsszavunk, ami mögött tábla, illetve táblák állhatnak. *(A táblát tágan kell értelmeznünk, mert nemcsak fizikai, létező tábláink lehetnek, hanem táblatípusú változóink, valamint táblaértéket visszaadó függvényeink, beágyazott lekérdezéseink is.)*

Ha táblát „szólítunk meg", mindig gondoljunk arra, hogy a táblák neve négytagú, és adott esetben ezt ki is kell írni *(például elosztott lekérdezések esetén)*.

A táblákra is vonatkozik az AS-os átnevezés, aminek nagy hasznát fogjuk venni többtáblás lekérdezéseknél, mert nemcsak rengeteg gépeléstől kímél meg minket, de egyes esetekben az egyértelműség meg is kívánja, hogy táblákat röptében átnevezzünk *(pl. önhivatkozó JOIN)*. A többtáblás lekérdezésekre külön fejezetben térünk vissza.

Így kérdezünk le függvényből:

```
SELECT * FROM dbo.nincsilyenfuggveny()
```

Így kérdezünk le beágyazott lekérdezésből, ahol a FROM után egy tetszőleges SELECT áll zárójelben, ettől táblává alakul:

```
SELECT * FROM (SELECT 'Akármi' as Mezo) as Tabla
```

3.4 A WHERE feltétel. Szűrések egyenlőségre, egyenlőtlenségre

Következő áldozatunk a szűrésekért felelős WHERE kulcsszó. Ez olyannyira fontos, hogy azt mondanám, az a SELECT, amelyikben nincs WHERE feltétel, nem érdemli meg, hogy lefuttassuk. Ha valaki átolvassa az ANSI SQL kritikáit, az egyik – többek között – az, hogy túl könnyű vele sokmillió sort lekérdezni, túl könnyű véletlenül az összes sort kitörölni vagy módosítani, lévén hogy a WHERE nem kötelező része a lekérdezésnek.

Ha teljesítménytuningolásról beszélünk, nem szabad figyelmen kívül hagyni, hogy a legnagyobb lehetőség mindig az, ha a kliensalkalmazások nem kérnek le egymillió sort feleslegesen, hogy abból kliensoldalon válasszák ki azt az ötöt, amit megjelenítenek a képernyőn. Sajnos még mind a mai napig számtalan ilyen buta alkalmazás létezik, sőt, jó pénzért árulják ezeket a piacon.

Mi ne legyünk ennyire bénák, vizsgáljuk meg, hogyan lehet a számunkra szükséges sorokat lekérdezni! A legegyszerűbb eset, amikor egy mezőértékre keresünk, például

```
WHERE Mezo=42
```

Túl azon, hogy a 42 a válasz a mindenre, kérdés, hogy vajon egy ilyen lekérdezés hány sort adna vissza abból a táblából, amelyikben a Mezo nevű mező szerepel? Sajnos erre nem az a válasz, hogy egyet. Ez ugyanis attól függ, mi ez a mező? Ha az egyedi kulcs, akkor egyet. Ha a népesség életkora, akkor – talán – a sorok 10%-át. Ha ez a negyvenkettesek táblája, akkor pedig az összes sort.

Természetesen van egyenlőtlenségvizsgálat *(<, >, >=, <=)* is, meg „nemegyenlő” *(<>)*, meg vannak logikai kifejezések *(AND, OR, NOT)*, de ezek mindenkinek a könyökén jönnek ki, hát menjünk is tovább a kevésbé hétköznapi szűrések felé!

3.4.1 BETWEEN

A BETWEEN egy alsó és egy felső értékhatár közé eső elemeket fogja meg. A BETWEEN 8 AND 9 értelemszerűen a 8 és 9 közé eső számokat adja vissza, beleértve a 8-at és a 9-et is. Nem is ez a lényeg, hanem a művelet univerzalitása. A BETWEEN ’2001.01.01’ AND ’2013.11.11’ pontosan jól működik dátummezők esetén, megspórolva ezzel egy csomó dátummatematikázást. Mi több, szöveges adatokon is működik! Vajon az alábbi lekérdezés visszaadja-e Gipsz Jakabot?

```
SELECT * FROM Customers
WHERE Lastname BETWEEN 'F' AND 'K'
```

Igen, ez a lekérdezés az elvárásainknak megfelelően működik, és egy csomó felesleges vergődéstől megkímélt minket.

3.4.2 LIKE

Ha már a szöveges mezőknél tartottunk, végezzünk el egy-két szöveges keresést! A LIKE operátor lehetővé teszi, hogy tartalomtöredékekre keressünk. *(Ez nem keverendő össze a szabadszöveges, magyarul freetext lekérdezésekkel, mert az teljesen másképp működik!)* Így kereshető meg az összes M-betűvel kezdődő vezetéknevű ügyfelünk:

```
SELECT * FROM Customers
WHERE Lastname LIKE 'M%'
```

A példából mindenki rájöhetett, hogy a % jel a dzsókerkarakter. Bárhol lehet használni a szöveges mezőben, tehát akár olyanokat is kereshetünk, akik zsó-ra végződnek:

```
SELECT * FROM Customers
WHERE Lastname LIKE '%zsó'
```

Ez utóbbi lekérdezéssel azonban vigyáznunk kell. Az SQL Server csak azoknál a kereséseknél tudja használni a szöveges indexeket, ahol a kifejezés balról zárt, ami emberi nyelven kifejezve azt takarja, hogy ahol nem bal szélen van a dzsóker karakter. Miért? Gondoljunk csak bele, hogyan találnánk meg a vezetéknevek szerint pedánsan sorba rendezett telefonkönyvben az összes olyan vezetéknevet, amelyik 'zsó'-ra végződik? Ugye, hogy szánalmasan, végiglapozva az egész telefonkönyvet? Csodák nincsenek. Ami szerint nincs sorban egy adat, a szerint az SQL Server nem tud okosan lekérdezni, csak a mező összes értékének végignyálazásával.

Ez nem jelenti azt, hogy ilyen lekérdezéseket ne használjunk, mert ha az alkalmazás logikája úgy kívánja, használnunk kell. Csak legyünk tudatában a háttérben zajló eseményeknek! *(Esetleg fontoljuk meg a Freetext indexek bevetését – bár a magyar nyelvvel nem tud mit kezdeni.)*

3.4.3 IN

Ha a lekérdezés egy kupac adat bármelyikének megléte esetén adjon vissza sorokat, sok-sok OR művelet helyett érdemes bevetni az IN operátort – ami valójában OR-okra fejti ki a lekérdezésünket. Ha valaki vagy Gipsz, vagy Fóti vezetéknévvel rendelkezik, így kérdezetjük le:

```
SELECT * FROM Customers
WHERE Lastname IN ('Gipsz', 'Fóti')
```

Valójában az IN mindenféle halmazra működik, így tehát az alábbi beágyazott SELECT-re is:

```
SELECT * FROM Customers WHERE Lastname IN
(SELECT DISTINCT Lastname FROM Customers WHERE FirstName LIKE
'Jakab')
```

Ez a lekérdezés összeszedi az összes vevőt, akinek a vezetékneve megegyezik másik olyan vevők vezetéknevével, akik Jakabok.

3.5 TOP

A TOP szócskára még térjünk vissza a teljesség kedvéért! Nemcsak a legelső x darab, hanem a legelső x százaléknyi sort is lekérdezhetjük vele a következő módon:

```
SELECT TOP 10 PERCENT * FROM Customers
```

Mi több, a lekérdezett sormennyiség egy kifejezés, amit szabadon kitölthetünk akár egy beágyazott lekérdezéssel, így ni:

```
SELECT TOP (SELECT COUNT(*) FROM Customers WHERE FirstName LIKE
'Jakab') PERCENT * FROM Customers
```

Ez ugyan elég ügyefogyott lekérdezés, mert annyi százaléknyi sort kérünk, ahány darab Jakabunk van, de legalább látjuk, hogy az értelmetlen lekérdezések írásának csak a fantáziánk szab határt.

3.6 DISTINCT

Az imént már használtuk, írjuk tehát le, mire is való. A DISTINCT kulcsszó segítségével kidobálhatjuk az azonos sorokat, egyetlen sort megtartva közülük. A művelet a teljes sorra vonatkozik, azt figyeli, ezért érdemes a mezők számát minimalizálni a lekérdezésben, vagy legalább az egyedi kulcsot kihagyni, mert ha a kulcsmező belekeveredik, értelemszerűen minden sor különböző lesz, a DISTINCT pedig nem csinál semmit. Egy oldallal korábban láthattunk egy gyönyörű példát a különböző vezetéknevek kigyűjtésére.

3.7 ORDER BY

A sorba rendezés nagy tudomány. Ennek az az oka, hogy - a változó hosszúságú szöveges mezők miatt - a rekordoknak nincs előre elrendelt fizikai sorrendjük, ezért ha valaki ORDER BY nélkül futtat le egy lekérdezést, ne csodálkozzon, hogy a következő adatmódosítás után esetleg más sorrendben kapja vissza a rekordokat, mint annakelőtte. Az ORDER BY maga a megoldás, az életbiztosítás, hogy biztosak lehessünk abban, hogy amit lekérdezünk, az mindig adott sorrendben érkezik.

Ami egyébként lehet növekvő és csökkenő is. Ha valaki le szeretné válogatni az öt legbecsesebb ügyfelét, akkor nyilván Amount szerint csökkenő sorrendben kell kérnie a listát, hogy a tetején legyenek a nagyobb értékek. A sorba rendezés irányát az ORDER BY Mező után biggyesztett ASC, DESC kulcsszavakkal módosíthatjuk kívánalmainknak megfelelően.

```
SELECT TOP 5 * FROM Transactions
ORDER BY Amount DESC
```

Több mezőre kiterjedő sorba rendezéseket is kérhetünk, egyszerűen vesszővel felsorolva a további kívánalmainkat az ORDER BY után, így ni:

```
SELECT TOP 5 * FROM Transactions
ORDER BY Amount DESC, CreationDate ASC,
Partner DESC
```

Ezenfelül az ORDER BY arra is képes, hogy a sorba rendezési oszlopokat pozíció alapján adjuk meg, valahogy így:

```
SELECT TOP 5 * FROM Transactions ORDER BY 1
```

Ilyenkor a lekérdezés eredményhalmazában legelső oszlopban található értékek alapján végzi a sorba rendezést.

Ennek az utóbbi szintaxisnak egyetlen helyen láttam gyakorlati hasznát: hekkelésnél. Micsoda szép átvezetőmondat a következő fejezetre! Magam sem tudtam volna szebben leírni, huh!

4 SQL Injection

SQL Server esetén az alábbi hekkelési lehetőség kínálkozik sok-sok alkalmazásnál: ha az alkalmazások olyan után vannak megírva (sok esetben így van), hogy gondolkodás nélkül beveszik az adatot a kliensprogramból, és belehelyettesítik egy SQL-lekérdezésbe, igen furcsa parancsok születhetnek.

Képzeljük magunk elé az alábbi *(ostoba ennek ellenére mégis gyakori)* bejelentkező-rutint, mely mondjuk egy PHP-alkalmazásban csücsül, és minden bejelentkezéskor lefut:

```
"SELECT * FROM Users
WHERE Username='" + $UName +"'"
AND Password=" + $Pwd +"'"
```

Ez úgy működik, hogy a bejelentkezéskor a paraméterek behelyettesítésével összeáll a lefuttatandó SQL-parancs, beküldjük az SQL Servernek, és ha megfelelő nevet és jelszót adunk meg, van eredményhalmaz *(jó eséllyel egy sort kapunk vissza)*, ha viszont bármelyik paraméter, akár a név, akár a jelszó rossz, üres halmaz az eredmény. Erre építve a programozó készíthet egy elágazást, hogy vagy beenged, vagy nem enged be az alkalmazásba. Hibátlan, ugye?

No, akkor adjuk be nekik a következő „felhasználónevet”: Jakab’ *(Jakab és egy aposztróf).* Ezzel egészen biztosan megborítjuk a kódot, mert innentől az aposztrófok száma páratlan, hiszen behoztunk egy plusz feleslegeset a képbe. Egészen pontosan ez lesz a lefuttatandó kód a behelyettesítések elvégzése után:

```
SELECT * FROM Users
WHERE Username='Jakab'' AND Password=''
```

A hibaüzenet egyértelmű: bezáratlan macskaköröm található a parancsban. Márpedig ilyen felhasználónév angol nyelvterületen előfordul. Kedvenc példám a híres kefegyáros, O’kefee. Vagy a celeb autóversenyzőt is említhetném, O’Rally.

Ha tovább ütjük a névmezőt, végül is ki tudunk keveredni a csávából, hiszen ha beviszünk neki egy jól irányzott kommentet, így ni: Jakab’—*(a sor végéig történő kommentelés jele a duplamínusz)*, a lekérdezés hirtelen megjavul, csak nem „úgy”, hanem „emígy”:

```
SELECT * FROM Users
WHERE Username='Jakab'--' AND Password=''
```

Figyeljük meg, hogy a jelszó zölddé vált, kikommentelődött, immár nem vesz részt a bejelentkeztetés folyamatában! Na még egy okos loginnevet neki, és be is jelentkeztünk. Legyen a nevünk mondjuk Jakab’ OR 1=1—

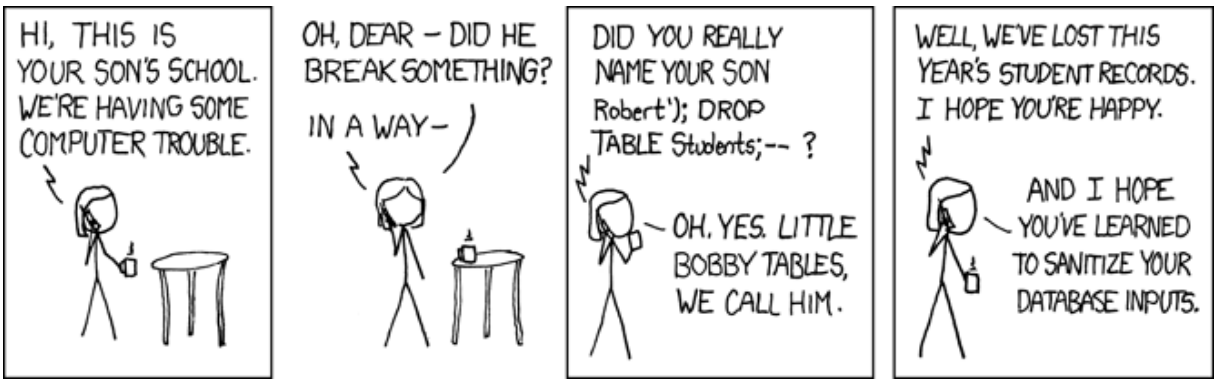
Behelyettesítve:

```
SELECT * FROM Users WHERE Username='Jakab'
OR 1=1--' AND Password=''
```

Futtassuk csak le fejben! Hány sort ad vissza? Ez most már csak attól függ, hogy az OR 1=1 milyen gyakran igaz. A loginnév nem játszik, mert ha a második feltétel igaz, a logikai VAGY művelet szabályai szerint az egész kifejezés igazzá válik. Nos, milyen gyakran igaz az 1=1? Mindig. Tehát hány sort ad vissza a lekérdezés? Mindet. A mind megfelel a bejelentkezés feltételeinek? Meg. Isten hozott az ingyenes előfizetések világában.

Innen már egyenes út vezet tetszőleges parancsok lefuttatásához. Ha a bejelentkező személyt úgy hívják, hogy Jakab' SHUTDOWN --, akkor szegény sosem tud bejelentkezni, mert az SQL Server az ő neve láttán mindig leáll.

Ha pedig valakit úgy hívnak, hogy Jakab'—DROP TABLE Students, nos, akkor behívják a szülőt elbeszélgetésre az iskolába.



17. ábra - Róbert meghekkeli az adatbázist

Ha egy autó rendszáma netán 'XYZ-123 DROP TABLE Cars--, akkor bizony a belügy felkötheti a gatyáját.

18. ábra - hogyan veszett el a személygépkocsi adatbázis egy rutinellenőrzés során?

Mielőtt az ORDER BY hekkerképességeit elemeznénk, még írjuk le, mi lehet a védekezés módja az SQL Injection ellen. Soha ne csináljunk a fentihez hasonló stringkolbászolást. Pont. Részletesebben: soha ne csináljunk olyat, hogy egy paraméter nem is paraméter, csak egy darabka string, amit befűzünk a parancsba! Hiába kényelmes, ne csináljunk ilyet! Helyette használjunk valami normális

paraméterbehelyettesítési módszert, mondjuk a .net megfelelő objektumait, vagy hívjunk meg tárolt eljárást a bejelentkeztetéshez!

4.1 Sorok letapogatása

A tetszőleges parancs átvezet minket a tetszőleges adat lekérdezése felé. Nincs olyan, hogy kicsi SQL Injection. Ha a rés megvan, azon keresztül bármit ki lehet húzni. Az első lépés persze a piszkálható tábla szerkezetének lekérdezése, hogy tudjuk, mi is hullott az ölünkbe.

Nos, az ORDER BY picit segít a tábla szerkezetének felderítésében. Az alábbi módszerrel egy támadó ki tudja tapogatni a visszaadott mezők számát. Beír hasra egy ORDER BY 60 sorba rendezést, ami elszáll, ha nincs 60 oszlopunk. 60-tól egyesével csökkentgetve végül el lehet jutni addig a számig, amikor már a lekérdezés rendben lefut – na pont annyi mezőnk van!

Nem célom az SQL Injection részletes taglalása, innen nem megyünk tovább. Az SQL Injection nagyon szép, és egyben nagyon bonyolult tudomány. Nagyon szép eredményeket lehet vele elérni, gondoljunk itt sok szeretettel a sitten ülő Albert Gonzalezre[11], aki 300 millió hitelkártyaadatot lopott el *(egymaga!)* SQL Injection segítségével! Az iménti iskolapélda arra alkalmas, hogy bemutassa azt a fajta „out of the box” gondolkodásmódot, ami a hekkerek sajátja.

Remélem, elég rémisztő ahhoz, hogy minden adatbázis-programozó egyszer s mindenkorra komolyan vegye az SQL Injectiont.

[11] http://en.wikipedia.org/wiki/Albert_Gonzalez

5 Data Query Language II. Csoportosítás, összegzés

Eljött az ideje a számításoknak. Ebben a fejezetben különböző összegzéseket fogunk végezni óriási mennyiségű adatunkon.

5.1 Aggregátumfüggvények

Az SQL Server számos hasznos és jól ismert aggregátumfüggvényt tartalmaz, valamint jó sok olyat, amiről ennyi év után sem tudom, hogy mire jó. Amikről könnyűszerrel meg tudjuk állapítani, hogy mit is csinálhatnak, azok a következők: AVG(), SUM(), MIN(), MAX(), COUNT() – és ezzel körülbelül ki is merült az általam értelmezhető függvények köre. Van még standard deviancia, meg ilyenek, de ezeket tudtommal mentális betegségekkel lehet inkább összefüggésbe hozni *(deviancia, ugye ☺)*. Kár hogy csak a devianciának van függvénye, mert ha a skizofréniának is lenne, azt mondhatnánk, az SQL Server teljes körűen alkalmas mentálhigiénés problémák megoldására. De így nem.

Az a néhány függvény viszont, amit értünk, pont azt csinálja, amire számítunk. Például a SUM() – meglepő módon – összegez. Gyorsan összegezzük tehát a Transactions táblában a pénzeket, hogy lássuk, hogyan is működik:

```
SELECT SUM(Amount) FROM Transactions
```

Közel zseniális, csak az vele a problema, hogy összeadta az almát a körtével, hiszen különböző pénznemekben tárolják a vagyonukat az ügyfeleink. Nosza, tegyük még bele a lekérdezésünkbe a pénznemet így, és lássuk, mire jutunk:

```
SELECT Currency, SUM(Amount) FROM Transactions
```

Egy rendkívül szép hibaüzeneten kívül túl sok eredményt nem kaptunk. A hibaüzenet így szól:

```
Msg 8120, Level 16, State 1, Line 1
Column 'Transactions.Currency' is invalid in the select list because it is not contained in
either an aggregate function or the GROUP BY clause.
```

Hirtelen fordításban: a Currency mező érvénytelen, mert nem szerepel összegzésben *(hát már hogy szerepelne?)*, de még a GROUP BY listában sem. Még magyarabbul: az a baj, hogy volt egy csomó sorunk, amit az Amount mezőre rápakolt SUM() egy sorrá rántott össze. A Currency mezővel meg nem csináltunk semmit, tehát jelenleg többsoros. Úgy kellene szegény SQL Servernek egy jó kis táblázatot faragnia, hogy az egyik mező többsoros, a másik meg egysoros. Nem fog menni.

5.2 Csoportosítás, GROUP BY

A hibaüzenetben hivatkozott GROUP BY úgy oldja fel ezt az ellentmondást, hogy az ott felsorolt mezők szerint csoportokat képez az eredményhalmazon, majd a SUM() is az egyes csoportokon dolgozik. Ha tehát az utasításnak engedelmeskedve betesszük a Currency mezőt a GROUP BY záradékba, csodálatos és – végre - értelmes eredményt kapunk:

```
SELECT Currency, SUM(Amount) FROM Transactions
GROUP BY Currency
```

	Currency	(No column name)
1	EUR	2370000000,00
2	HUF	888888,00
3	USD	8768700004,00

19. ábra - első csoportosított összegző lekérdezésünk eredménye

Ez nagyon jól sikerült, ezen felbuzdulva jelenítsük meg a partnert is a lekérdezésben! Mondjuk rakjuk a csoportosítási feltételek közé! Merthogy lehet egynél több feltétel szerint csoportosítani, bizony!

```
SELECT Currency, Partner, SUM(Amount)
FROM Transactions
GROUP BY Currency, Partner
```

	Currency	Partner	(No column name)
1	EUR	Enron	2370000000,00
2	USD	Fuggers International	8768700000,00
3	USD	Lehman Brothers	0,00
4	HUF	Postabank	888888,00
5	USD	Uncle Sam	4,00

20. ábra - a Lehman Brothers lebuktatása

Még hivatalosabb lenne a Lehman Brothers csődje, ha csak a zéró összegű sorokat adná vissza a lekérdezésünk. Ha megpróbálkozunk a WHERE-feltétellel, és oda beírjuk, hogy WHERE SUM(Amount)=0, hibaüzenetet kapunk, hogy nincs ilyen sor. Valóban, amikor a WHERE kiértékelődik, még nincs. Ezt követi a GROUP BY és az összegzés, és ennek a második táblának van egy második, saját WHERE feltétele, úgy hívják, HAVING.

5.3 A második WHERE - a HAVING

Kérdezzük le a zéróra játszó partnereket!

```
SELECT Currency, Partner, SUM(Amount) FROM Transactions
GROUP BY Currency, Partner
HAVING SUM(Amount)=0
```

	Currency	Partner	(No column name)
1	USD	Lehman Brothers	0,00

21. ábra - teljes csőd...

Ez remekül működik. Ne ijedjünk meg attól, hogy most a lekérdezésben kétszer szerepel a SUM() függvény, az SQL Server elég okos ahhoz, hogy csak egyszer számolja ki az összegzést.

5.4 Részösszegek készítése, ROLLUP

A főnökünk előállt azzal az ötlettel, hogy jó-jó ez a csoportosítás, de ő mégiscsak szeretné látni a pénznemenkénti bontatlan összegeket is. Mit tehetnénk? Engedelmeskedünk neki. A cél érdekében beírjuk az előző parancs végére, hogy WITH ROLLUP:

```
SELECT Currency, Partner, SUM(Amount)
```

```
FROM Transactions
GROUP BY Currency, Partner WITH ROLLUP
```

Hát, ez az eredmény már nem beszél önmagáért, ezért pirossal kiemeltem, milyen új adatok jelentek meg benne:

	Currency	Partner	(No column name)
1	EUR	Enron	2370000000,00
2	EUR	NULL	2370000000,00
3	HUF	Postabank	888888,00
4	HUF	NULL	888888,00
5	USD	Fuggers International	8768700000,00
6	USD	Lehman Brothers	0,00
7	USD	Uncle Sam	4,00
8	USD	NULL	8768700004,00
9	NULL	NULL	11139588892,00

22. ábra - a WITH ROLLUP eredménye

Mindegyik új sornak tulajdonsága, hogy van benne egy vagy több NULL érték. Ezek azért vannak, mert a ROLLUP ténykedése valójában az, hogy megcsinálja a csoportosításokat úgy, ahogy előzőleg is, majd beszúr egy olyan részösszeget, ahol az utolsó csoportosítást nem veszi figyelembe (ez nálunk a Partner), oda tehát egy NULL-t tesz. Ezzel megmagyaráztuk az első három piros sort: pénznemenként egy-egy totál. Na de mi a legvégén a NULL-NULL? Az pedig a grand totál!

A ROLLUP jobbról haladva „felgöngyölíti” a csoportosításokat, és egészen addig teszi ezt, amíg az összes csoportosítástól megszabadul, ergo az összes összevont csoportmezőbe NULL kerül, az összegmezőbe pedig egy orbitális grand totál.

Itt jön be a képbe a csúnya NULL megint, mert ugyan hogyan tudjuk most megállapítani, hogy egy-egy NULL valódi NULL-e, mert NULL pénznem is volt az adatbázisban, vagy egy a lekérdezés által-generált NULL? Jó hírem van, külön függvény dolgozik a ROLLUP mellett, ami pont ezt hivatott kimutatni, a neve GROUPING(), és így néz ki a használata:

```
SELECT Currency, GROUPING(Currency),
Partner, GROUPING(Partner), SUM(Amount)
FROM Transactions
GROUP BY Currency, Partner WITH ROLLUP
```

	Currency	(No column name)	Partner	(No column name)	(No column name)
1	EUR	0	Enron	0	2370000000,00
2	EUR	0	NULL	1	2370000000,00
3	HUF	0	Postabank	0	888888,00
4	HUF	0	NULL	1	888888,00
5	USD	0	Fuggers International	0	8768700000,00
6	USD	0	Lehman Brothers	0	0,00
7	USD	0	Uncle Sam	0	4,00
8	USD	0	NULL	1	8768700004,00
9	NULL	1	NULL	1	11139588892,00

23. ábra - ott mosolyognak a GROUPING() bitek

Ha ez megvan, fel is vagyunk készülve valami nagyon vad dologra. A főnök azt kéri, hogy ábrázoljuk az egyes pénznemekhez és partnerekhez kiszámolt részösszegeket egy Descartes-koordinátarendszerben. *(A főnök vagy matematikus, vagy nem felejt. Végül is általános iskolában foglalkoztunk vele. Vagy nem?)*

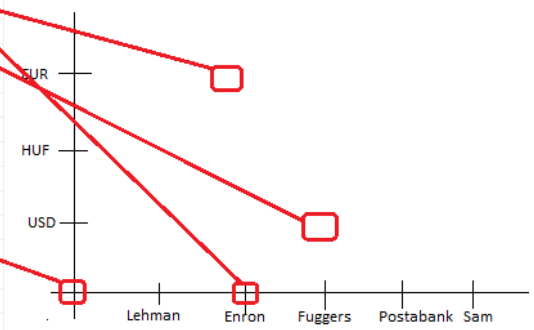
5.5 Kockulat, CUBE

Mielőtt az elkeseredettség úrrá lenne a lelkünkön, elmondom, hogy a főnök kérésének igen egyszerűen eleget lehet tenni. Csupán a ROLLUP szót kell kicserélni CUBE-ra, és készen is vagyunk. Hogy miért? Mert a CUBE kvázi ugyanazt csinálja, mint a ROLLUP, csak nem jobbról balra, hanem minden kombinációban kiszámolja a részösszegeket, amelyeket ezek után a GROUPING() függvény értékei alapján el tudunk helyezni a koordinátarendszerben.

```
SELECT Currency, GROUPING(Currency), Partner, GROUPING(Partner),
SUM(Amount) FROM Transactions
GROUP BY Currency, Partner WITH CUBE
```

Egy picivel több részösszeget kapunk, hisz most már minden létező és nem létező csoportosítási és nem csoportosítási kombinációra kiszámolódott, de annyi baj legyen, a lényeg, hogy értjük! (?)

	Currency	(No column name)	Partner	(No column name)	(No column name)
1	EUR	0	Enron	0	2370000000,00
2	NULL	1	Enron	0	2370000000,00
3	USD	0	Fuggers International	0	8768700000,00
4	NULL	1	Fuggers International	0	8768700000,00
5	USD	0	Lehman Brothers	0	0,00
6	NULL	1	Lehman Brothers	0	0,00
7	HUF	0	Postabank	0	888888,00
8	NULL	1	Postabank	0	888888,00
9	USD	0	Uncle Sam	0	4,00
10	NULL	1	Uncle Sam	0	4,00
11	NULL	1	NULL	1	11139588892,00
12	EUR	0	NULL	1	2370000000,00
13	HUF	0	NULL	1	888888,00
14	USD	0	NULL	1	8768700004,00



24. ábra - kétdimenziós kocka, de az is síkba terítve

Néhány értéket átpakoltam a koordinátarendszerbe, hogy látszódjon a logika. A grand totál kerül az origóba, minden kistotál az adott tengelyre, a kiszámított összegecskék pedig a megfelelő pozíciókra.

Azt is látni kell, hogy valójában ez egy kocka, csak a mi példánkban éppen kétdimenziós, mert mindössze két csoportosítási feltételünk van. Ha három lenne, rendes, 3D-s kockában tudnánk az adatokat ábrázolni, ha négy, akkor pedig négydimenziósban.

A rajzolgatáshoz sok sikert kívánok, de megjegyzem, ezt így nem szoktuk ábrázolni. Arra való az Excel, hogy ezeket az akárhány dimenziós kockákat emészthetővé, kezelhetővé tegye.

Nem tudom, feltűnt-e, hogy rövid kirándulást tettünk az OLAP-kockák világába?

5.6 Csoportosítás extrém kombinációkban

És ezzel még mindig nincs vége. Mind a ROLLUP, mind a CUBE kötöttpályás jármű, tetszőleges összegzéskombinációkra nem képes. Erre is van azonban lehetőség, sőt, meglepő módon igény is. A

YouTube-on található TSQL-ámokfutás[12] című videómban látható egy példa, aminél a GROUPING SETS lehetőség nagyban megkönnyítette egy komplex lekérdezés kialakítását.

Íme egy példa, amikor csak és kizárólag a Currency és a Partner részösszegeire vagyok kíváncsi, a pici totálokra egyáltalán nem:

```
SELECT Currency, GROUPING(Currency), Partner, GROUPING(Partner),
SUM(Amount)
FROM Transactions
GROUP BY GROUPING SETS((Partner), (Currency))
```

És az eredmény:

	Currency	(No column name)	Partner	(No column name)	(No column name)
1	EUR	0	NULL	1	2370000000,00
2	HUF	0	NULL	1	888888,00
3	USD	0	NULL	1	8768700004,00
4	NULL	1	Enron	0	2370000000,00
5	NULL	1	Fuggers International	0	8768700000,00
6	NULL	1	Lehman Brothers	0	0,00
7	NULL	1	Postabank	0	888888,00
8	NULL	1	Uncle Sam	0	4,00

25. ábra - teljesen egyedi összegzések kiválasztása

[12] http://www.youtube.com/watch?v=I6ILya-p-DE

6 Data Query Language III. Táblák összekapcsolása

Tökéletesen megtervezett adatbázisunk messze több, mint egy táblából áll. Ennek megfelelően lekérdezéseinknek is figyelembe kell vennie ezt a jellegzetességet, meg kell tanulnunk összekapcsolt táblákból lekérdezni adatokat.

A táblák összekapcsolását a JOIN operátor végzi, amely a FROM-ban felsorolt táblák között teremt kapcsolatot. Sokan – tévesen – azt hiszik, hogy lekérdezéskor tekintettel kell lennünk a táblaszerkezet kialakításakor megadott referenciális integritási szabályokra. Erről szó sincs. Annyi azonban igaz, hogy tábláinkat leggyakrabban a felállított apa-fiú kapcsolatok mentén kérdezzük le.

De hogy rögtön valami értelmetlennel kezdjük, a JOIN szintaxisát egy olyan lekérdezésen mutatom be, amelyik úgy kapcsolja össze a bankszámlákat a tranzakciókkal, hogy párt képez, ahol a bankszámla létrehozási dátumának hónapsorszámát kisebb, mint a tranzakció összegéből nyert 3%-os kártyadíj egész része. Csak hogy bizonyítsam, hogy bármit bármivel összekapcsolhatunk.

```
SELECT * FROM Accounts INNER JOIN Transactions
ON DATEPART(MONTH, Accounts.CreationDate) < CAST(CardFee AS INT)
```

Sőt, akár olyan feltételt is írhatunk a dzsoinhoz, amit a hekkerek szoktak használni:

```
SELECT * FROM Accounts INNER JOIN Transactions
ON 1=1
```

Szép kis eredményhalmazt kapunk, babonák és félelmek legyőzve.

6.1 A természetes JOIN, Equijoin

Figyeljük meg az alapszintaxist: FROM Table1 INNER JOIN Table2 ON feltétel! Ebben egyedül az INNER JOIN lehet ismeretlen, hát elmagyarázom. Az INNER, vagy más néven természetes, naturális JOIN egyezőséget keres az érintett táblákban a feltétel mentén. Pontosabban akkor ad sort a kimenetre, ha a feltétel igaz.

Apa-fiú esetben ez a sorpárokat eredményezi, mármint ha a feltételben egyenlőséget, hivatalosan equijoint használunk. Lássunk akkor egy értelmes példát is, keressük a tranzakciókhoz tartozó számlaszámot! A kapcsolat feltétele, hogy az Accountnak az AccountID-je megegyezzen a Transactionban megbújó apa-azonosítóval, aminek trükkösen szintén AccountID a neve:

```
SELECT AccountNumber, Transactions.*
FROM Accounts INNER JOIN Transactions
ON Accounts.AccountID=Transactions.TransactionID
```

Mivel túl hosszúak a sorok, elkezdünk rövidíteni. Először is az INNER kulcsszó elhagyható, mert a természetes JOIN az alapértelmezés. Emellett az összetett táblás kifejezésekben erőteljesen aliasolni szoktunk, hogy a hosszú táblaneveket ne kelljen annyiszor kiírni. Így áll elő a fenti lekérdezésből ez a vele azonos eredményt adó, de rövidebb változat:

```
SELECT AccountNumber, Transactions.*
FROM Accounts a JOIN Transactions t
ON a.AccountID=t. AccountID
```

És most néhány szót az eredményhalmazról. Ha nem vigyázunk, ez a lekérdezés még gyilkosabb lehet, mint az egy táblán elkövetett korlátozás nélküli lekérdezés. A JOIN ugyanis egy óriáslepedőt képez az egyesített táblákból, sokszorosan, gyakorlatilag minden gyermekrekordhoz megismételve az aparekordot. Egymillió gyerek öt olyan apával, akiknek fényképe van – brutális.

Csak a szokásos mondókámat sulykolhatom: legyen WHERE-feltételed tenéked.

Össze tudunk-e kapcsolni kettőnél több táblát egy még nagyobb lepedő előállításához? Természetesen. A JOIN további táblákkal úgy bővíthető, hogy az első kettőt mint lepedőt hozzákapcsoljuk a harmadikhoz, majd azt mint lepedőt hozzákötjük a negyedikhez és így tovább. Nekünk négy táblánk van, az ebből készült mindent betakaró lepedővászon így néz ki:

```
SELECT *
FROM Cities c
JOIN Customers cu ON c.CityID=cu.CityID
JOIN Accounts a ON a.CustomerID=cu.CustomerID
JOIN Transactions t ON a.AccountID=t.AccountID
```

És az eredmény, a 86 mezős, soksoros lepedő:

	CityID	Name	CustomerID	FirstName	LastName	CityID	AccountID	CustomerID	AccountNumber	CreationDate	URLCode	TransactionID
1	1	Budapest	1	Jakab	Gipsz	1	1	1	12345678-64353453	2012-02-26 18:24:50.8330000 +00:00	F76EF604-20F2-419D-999B-E3B036C3DFD8	1
2	1	Budapest	1	Jakab	Gipsz	1	1	1	12345678-64353453	2012-02-26 18:24:50.8330000 +00:00	F76EF604-20F2-419D-999B-E3B036C3DFD8	2
3	1	Budapest	1	Jakab	Gipsz	1	1	1	12345678-64353453	2012-02-26 18:24:50.8330000 +00:00	F76EF604-20F2-419D-999B-E3B036C3DFD8	3
4	2	Gödöllő	2	Benő	Kő	2	2	2	9876543-64353453	2012-03-05 01:37:24.0430000 +00:00	F71FA2E8-728C-4C36-8EDC-9148EED0B90A	4
5	1	Budapest	1	Jakab	Gipsz	1	1	1	12345678-64353453	2012-02-26 18:24:50.8330000 +00:00	F76EF604-20F2-419D-999B-E3B036C3DFD8	5
6	1	Budapest	1	Jakab	Gipsz	1	1	1	12345678-64353453	2012-02-26 18:24:50.8330000 +00:00	F76EF604-20F2-419D-999B-E3B036C3DFD8	6

De hova tűnt Damon Hill?

6.2 A "természetellenes" JOIN-ok, OUTER

A természetes JOIN társkereső típus. Akinek nincs társa, az nem is létezik. Sok alkalmazásnál fedezhető fel az a hiba, amit röviden csak az INNER JOIN átkának hívhatunk, hogy nem jelenik meg a pénztárgépen a termék, mert nem vittek fel hozzá semmilyen akciót. Nem jön fel a könyvtári katalógusban a szerző, mert egyetlen könyve sincs bent. Nem lehet jelentkezni a weben a tanfolyamra, mert a háttérben valaki nem rendelt hozzá tantermet - ami egyébként a regisztrációt semmilyen formában nem akadályozza. És persze hova tűnt Damon Hill?

Kell lennie olyan megoldásnak, amivel nem veszítjük el a bankszámlával még nem rendelkező ügyfeleket. Van is, úgy hívják, OUTER JOIN, és úgy viselkedik, hogy egy adott táblából megtartja az árva sorokat is. Pontosabban abból a táblából, amelyikre rábökünk, hogy na, ez marad. A bal *(LEFT OUTER JOIN)*, vagy a jobb *(RIGHT OUTER JOIN)*, vagy mindkettő *(FULL OUTER JOIN)*. Ez utóbbihoz elég beteg, referenciális integritást nélkülöző adattartalom is szükségeltetik…

Itt jön Damon Hill:

```
SELECT *
FROM Cities c JOIN Customers cu on c.CityID=cu.CityID
LEFT OUTER JOIN Accounts a ON a.CustomerID=cu.CustomerID
LEFT OUTER JOIN Transactions t ON a.AccountID=t.AccountID
```

	CityID	Name	CustomerID	FirstName	LastName	CityID	AccountID	CustomerID	AccountNumber	CreationDate
1	1	Budapest	1	Jakab	Gipsz	1	1	1	12345678-64353453	2012-02-26 18:24:50.8330000 +00:00
2	1	Budapest	1	Jakab	Gipsz	1	1	1	12345678-64353453	2012-02-26 18:24:50.8330000 +00:00
3	1	Budapest	1	Jakab	Gipsz	1	1	1	12345678-64353453	2012-02-26 18:24:50.8330000 +00:00
4	1	Budapest	1	Jakab	Gipsz	1	1	1	12345678-64353453	2012-02-26 18:24:50.8330000 +00:00
5	1	Budapest	1	Jakab	Gipsz	1	1	1	12345678-64353453	2012-02-26 18:24:50.8330000 +00:00
6	2	Gödöllő	2	Benő	Kő	2	2	2	9876543-64353453	2012-03-05 01:37:24.0430000 +00:00
7	2	Gödöllő	4	Damon	Hill	2	NULL	NULL	NULL	NULL

26. ábra - megvan Damon Hill!

Remek, itt van Damon Hill, de lyukas nemcsak a benzintartálya, hanem a kereke is. A gyermektáblákban minden adata NULL lett. Ami nem meglepő, ha tudjuk, hogy nincs is gyermektábla-bejegyzése, egy sem. Most ismét itt állunk egy sereg NULL-lal, és első látásra nem tudjuk eldönteni, hogy amit kaptunk, az igazi NULL *(mert ugyan van bankszámlaszáma Damon Hillnek, de nincs kitöltve egyik mező sem)*, vagy generált NULL *(mert a gyermekrekord egyáltalán nem létezik)*.

Erre a problémára van egy kézenfekvő megoldás. Ha figyelmesebben megszemléljük az eredményt, egy árulkodó ellentmondást fedezhetünk fel. Maga a lekérdezés egyenlőségen alapult, igaz? De ha megnézzük, a CustomerID mezőkre ez nem igaz! Az apatáblában 4, a gyermektáblában NULL. Ami úgy egyenlő, hogy közben nem egyenlő, az a generált NULL biztos jele. És onnantól abból a gyermektáblából minden NULL generált – nyilván.

A másik fontos dolog, hogy kétszer kellett kiírnom a LEFT OUTER JOIN-t. Ennek oka az, hogy bármelyik gyermektábla ismét kihajítja Damon Hillt, ha nincs benne odavonatkozó sor. Ebből az következik, hogy ha egy szinten elkezdtünk OUTER-ezni, azt következetesen végig kell vinni „lefelé” az adott irányban, különben nem csináltunk semmit.

6.3 Tesztadatgenerálás CROSS JOIN-nal

Van a JOIN-oknak egy különleges formája, a CROSS JOIN, ami az eddigiektől eltérően nem igényel feltételmegadást, mivel a két hivatkozott tábla minden sorát gondolkodás nélkül összepárosítja egymással. Én ezt általában tesztadatgenerálásra használom. Csinálok egy vezetéknevek és egy keresztnevek táblát, mindegyikben tíz-tíz névvel, majd összekrosszolom őket, és máris van 100 vevőm. Ha még középső nevet is felveszek tízet, és három táblát krosszolok, ezer vevőm lesz. Ezer vevőt ha összekrosszolok ezer egyéb adattal, egymillió soros táblát nyerek. És így tovább. Nem olyan nagy ördöngösség szert tenni egy több millió soros táblára.

6.4 Self Join

Nem külön jointípus, mégis külön tárgyalják az önhivatkozó kapcsolatokat. Egy olyan kapcsolatot képzeljünk el, ahol egy táblát önmagával kapcsolunk össze valamilyen feltétel mentén! Tipikus példa, amikor egy táblázatban valamilyen hierarchikus, például főnök-beosztott viszonyt képezünk le oly módon, hogy minden ember idegen kulccsal mutogat a főnökére. Egy ilyen táblában egy főnök beosztottjait úgy tudjuk előállítani, ha ezt a táblát önmagával összekapcsoljuk a hierarchiát megvalósító mező mentén.

Készítsünk egy táblát a bank dolgozóival! Ne nagyon cicomázzuk, elég, ha nevük van – meg persze a hierarchia:

```
CREATE TABLE Employees(
EmployeeID INT NOT NULL PRIMARY KEY IDENTITY,
BossID INT NULL FOREIGN KEY REFERENCES Employees(EmployeeID),
Name NVARCHAR(88)
)
```

Érdemes megfigyelni, hogy a BossID, mely a főnökre mutat, nullozható, máskülönben egy árva darab sort nem lehetne felvinni ebbe a táblába. *(Rejtvény: miért?)*

És akkor jöjjenek a dolgozók:

```
INSERT Employees(BossID, Name) VALUES
(NULL, 'Mérges Gyula'),
(1, 'Szorgos Kata'),
(1, 'Pancser Géza'),
(3, 'Okos Imre'),
(3, 'Lusta Disznó')
```

Mérges Gyula a főnök, mert az ő főnöke NULL – nincs főnöke. Két közvetlen beosztottja Szorgos Kata titkárnő és Pancser Géza műszaki főosztályvezető. Ez utóbbinak van két mérnöke, Okos Imre és Lusta Disznó.

Hogyan tudjuk lekérdezni Mérges Gyula beosztottait? Így:

```
SELECT * FROM Employees e1 JOIN Employees e2
ON e1.EmployeeID=e2.BossID
WHERE e1.EmployeeID=1
```

	EmployeeID	BossID	Name	EmployeeID	BossID	Name
1	1	NULL	Mérges Gyula	2	1	Szorgos Kata
2	1	NULL	Mérges Gyula	3	1	Pancser Géza

27. ábra - Mérges Gyula beosztottai

Természetesen a másik irányban is kutakodhatunk a hierarchiában. Vajon kicsoda Okos Imre közvetlen főnöke? A JOIN feltételének megfordításával felfelé ütünk:

```
SELECT * FROM Employees e1 JOIN Employees e2
ON e1.BossID=e2.EmployeeID
WHERE e1.EmployeeID=4
```

	EmployeeID	BossID	Name	EmployeeID	BossID	Name
1	4	3	Okos Imre	3	1	Pancser Géza

28. ábra - Okos Imre főnöke

Sajnos nincs tovább. Ha egynél több hierarchiaszintet szeretnénk egy lekérdezéssel feltárni, az Employees táblát újra és újra hozzá kellene még kapcsolnunk. Ami járhatatlan, hisz nem tudjuk, hány szintet is kell bejárni.

A hierarchiák kifejtésének sokkal kifinomultabb módja az úgynevezett Common Table Expression, amire ebben a könyvben nem térek ki annak bonyolultsága miatt. Rekurzió… Brrrrr… A másik lehetőség pedig, hogy nem idegen kulcsokkal, hanem a HierarchyID adattípus felhasználásával valósítjuk meg a hierarchiát. Erre az utolsó fejezetben látunk példát.

Evezzünk békésebb vizekre! A táblák összekapcsolásának másik módja, amikor egymással kompatibilis eredményhalmazokat fűzünk össze.

6.5 UNION, INTERSECT, EXCEPT

Gyakori helyzet, hogy egy erőteljesen használt táblából a régebbi rekordokat átemeljük egy másik, hasonló vagy azonos felépítésű táblába archiválás céljából. A régi adatok eltávolításával az élő tábla megkönnyebbül. Innentől azonban az adataink két táblában vannak, és ha pont egy olyan időszakaszt kell elemeznünk, ami mindkét táblát érinti, gondban vagyunk. Hogyan lehet két lekérdezés eredményét összefűzni? Hát UNION operátorral! Az alapműködést remekül szemlélteti az alábbi lekérdezés:

```
SELECT 1
UNION
SELECT 42
```

A két külön lekérdezés egy táblát, benne két sort eredményez. Mi a helyzet, ha a lekérdezést így módosítjuk:

```
SELECT 1
UNION
SELECT 1
```

A két lekérdezés …. öööö … egy sort eredményez?! Igen, ez így van. A UNION operátor alapértelmezés szerint magára vesz egy DISTINCT műveletet is. Ha meg szeretnénk őrizni a két táblából az azonos sorokat, UNION ALL-t kell használnunk.

Egy másik fontos odafigyelnivaló az egymás alá passzított mezők kompatibilitása. Ez:

```
SELECT 1
UNION
SELECT 'targonca'
```

elszáll. Almát a körtével nem lehet azonos oszlopba hozni. Azonban mint minden problémára, erre is van megoldás. Kivételesen a NULL lesz a mi barátunk, mert ugyan semmivel sem egyenlő *(még önmagával sem!)*, de mindennel kompatibilis! Ha a targoncát – nagy nehezen - eltolom egy mezővel jobbra, ahol a felső táblában esetleg egy NULL szerepel, a targonca be fog szépen sorolni alá, így ni:

```
SELECT 1, NULL
UNION
SELECT NULL, 'targonca'
```

	(No column name)	(No column name)
1	1	NULL
2	NULL	targonca

29. ábra - egymással összeegyeztethetetlen adatok UNION-olása

Hogy ez meg mire jó? Hát hekkelésre! Tételezzük fel, hogy hekkerünknek van egy frankó SQL Injection hozzáférése a Customers táblánkra, de neki az Accounts tábla adatai kellenének! Igen ám, de az a fránya lekérdezés, amit sérülékenynek talált, így néz ki, tehát nem beszélhető le a Customers tábláról:

```
"SELECT * FROM Customers
WHERE LastName = '" + Param + "'"
```

Ugyanakkor tetszőleges adatot hozzá lehet fűzni, UNION segítségével, csak arra kell ügyelni, hogy az új adatok velük kompatibilis mezők alá kerüljenek. Az AccuntNumber mondjuk például a LastName alá. Beadja tehát a weblap megfelelő mezőjét kitöltve, szabályosan ennek a boci lekérdezésnek a következő feltételt: Akárki' UNION SELECT NULL, NULL, AccountNumber, NULL FROM Accounts—

A lefutó lekérdezés pedig ez lesz:

```
SELECT * FROM Customers
WHERE LastName = 'Akárki'
UNION
SELECT NULL, NULL, AccountNumber, NULL FROM Accounts--'
```

És máris az övé 300 millió bankkártyaadat. Én mondtam, hogy lekérdezési stringet nem fűzünk! Ugye megmondtam!

6.6 Beágyazott lekérdezések

Beágyazott lekérdezést már korábban használtunk. Ők azok a zárójelbe tett SELECT utasítások, amelyek egy-egy kifejezés helyén állhatnak, és külön, előre kiértékelődnek. Visszatérési értékük szerint megkülönböztetünk egy- és többértékű beágyazott lekérdezéseket. Az egyértékűeket minden teketória nélkül felhasználhatjuk például egyenlőségvizsgálat esetén például így: WHERE Mezo=(SELECT 1ertek FROM Tablacska). Ha azonban megcsúszik a beágyazott lekérdezés, és egynél több értéket produkál, az iménti egyenlőség úgy borul fel, hogy meg sem áll a hibaüzenetig. Van azonban megoldás a problémára. WHERE feltétel esetén az alábbi módosítókat használhatjuk beágyazott lekérdezések és egyenlőségfeltétel összeboronálására:

- ANY: a beágyazott lekérdezés eredményhalmazából ha bármelyik kielégíti a feltételt, oké
- ALL: a beágyazott lekérdezés eredményhalmazából ha mindegyik egyszerre kielégíti a feltételt, oké
- SOME: ez ugyanaz, mint az ANY, ANSI SQL nyelven megfogalmazva

Példa:

```
SELECT * FROM Customers
WHERE LastName = ANY (SELECT DISTINCT Lastname FROM Customers WHERE
FirstName LIKE 'Jakab')
```

Ez a korábbi IN-es példa átírva ANY-re, és továbbra is azokat a vevőket adja vissza, akiknek a vezetékneve megegyezik bármelyik másik Jakab vezetéknevével.

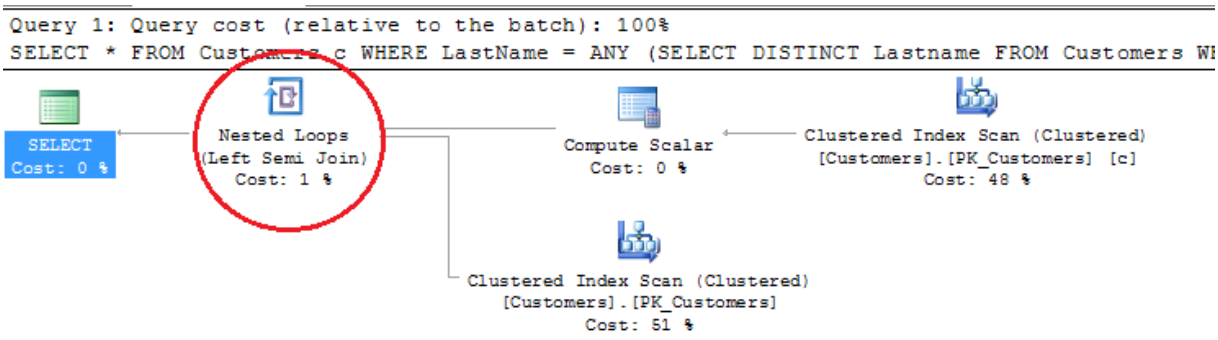
6.7 Korrelált "szabkveri"

Eddig mindig úgy vettük, hogy a beágyazott lekérdezés önmagában elvan, külön, önmagában jó előre lefuttatható, produkál egy értéket, amit behelyettesítünk és kész. Van azonban a beágyazott

lekérdezéseknek egy formája, amelyik nem ilyen. Lehetőség van ugyanis arra, hogy a hívó belepiszkáljon a hívott lekérdezésbe. Ezeket hívjuk korrelált lekérdezéseknek. Jellegüket tekintve a JOIN-okkal rokonok. Ha például az előző lekérdezést átalakítjuk úgy, hogy figyelembe vesszük a hívó keresztnevét, egy olyan lekérdezéshez jutunk, mintha a Customers táblát önmagához JOIN-oltuk volna a FirtName mező mentén. *(A külső Customer táblának kénytelen voltam becenevet adni, hogy megkülönböztessem a belsőtől.)*

```
SELECT * FROM Customers c
WHERE LastName = ANY (SELECT DISTINCT Lastname FROM Customers
WHERE FirstName LIKE c.FirstName)
```

Ennek a lefutása igen érdekes jelleget mutat. Mindaddig nem lehet kiértékelni a belső lekérdezést, amíg meg nincs a külső következő sora. Így ez most külső-belő-külső-belső lüktetéssel fog lefutni, soronként. Illetve… az SQL Server felismeri, hogy ez olyan, mint egy JOIN, és át is írja a lekérdezési tervet JOIN-ra. Kis okos!



30. ábra - beágyazott lekérdezés, amiből a háttérben JOIN lett

6.8 Nézetek

A beágyazott lekérdezések külön típusa az elmentett beágyazott lekérdezés, a VIEW, mely nem más, mint egy névvel ellátott, elmentett SELECT, amit később táblaként tudunk felhasználni.

A nézet táblaként viselkedik a hívó számára, de egy olyan tábla, ami az igazi fizikai tábláknak csak egy általunk választott szeletét mutatja meg. Néhány sort és néhány oszlopot.

Nézettel el lehet rejteni bizonyos nem publikus oszlopokat a kíváncsi tekintetek elől. Néhány gyakorlati példa:

- A felhasználó ne lássa a naplózási oszlopokat. Ne is tudjon róla, hogy vannak „ki_vitte_fel” meg „mikor_tetted_ezt” oszlopok
- Ha egy adatbázisban logikai törlést valósítunk meg a sorokon egy isDeleted BIT típusú mezővel, ennek nem szabad látszania az éles megjelenítésben. Sőt, ugyanezzel a bittel könnyedén tudunk csinálni egy Kuka nézetet, ami csak a törölteket mutatja.
- Minden utazó ügynök csak azokat az eladásokat lássa, amelyek őhozzá tartoznak!
- stb.

Emellett a nézetek arra is alkalmasak, hogy komplexitást rejtsünk el vele. Megfogjuk a háromtáblás JOIN-t, és belerakjuk egy nézetbe, innentől a hívók számára ez csak egyetlen hatalmas tábla.

Nem beszélve a számításigényes GROUP BY és aggregátumlekérdezésekről, amelyeket szintén bele lehet pakolni egy nézetbe, hogy soha többé ne lássuk a gusztustalan sok kódot.

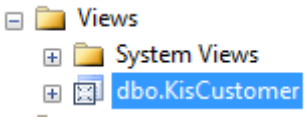
És végül a nézetek tartalmát le is lehet mentetni az SQL Serverrel, aminek hatására tizenmilliószorosra gyorsulnak, mert valójában nem futnak le többé, hanem a lemezről felolvassák a kész adatokat.

Az alábbi példában egy kompelxitáselrejtő nézetet látunk, amely nem engedi láttatni a végfelhasználóval, hogy a Customer táblának még mindenféle karbantartási mezői is vannak (URLCode például).

```
CREATE VIEW KisCustomers AS
SELECT CustomerID, LastName, FirstName
FROM Customers
```

A CREATE VIEW törzse egyetlen, bár *(majdnem)* tetszőlegesen bonyolult SELECT lehet[13].

A kész nézet beül a fába:



31. ábra - a nézet az Object Explorerben

Használni pedig úgy lehet, mintha ő maga is tábla volna:

```
SELECT * FROM KisCustomers
```

Ha tábla, akkor természetesen írható is. Igen, a KisCustomerre rá lehet engedni INSERT, UPDATE és DELETE utasításokat, amelyeket az ésszerűség határán belül el is végez. Ez a gyakorlatban annyit jelent, hogy ha az alatta lévő sorban van egy mező, mely nem látszik a nézetben, nem NULL-ozható és alapértelmezett értéke sincs, akkor sehonnan sem kap értéket, és az INSERT elszáll. Arra kell tehát figyelni a nézetek módosítása során, hogy ami nem látszik, az is van, és befolyásolja szerkesztési ténykedésünket. Van, de nem látszik != nincs.

[13] A nézet törzsében megadott lekérdezésre bizonyos korlátozások vonatkoznak, amit majd ki-ki a hibaüzenetek mentén jól kitapasztal. Például nem lehet a SELECT-ben ORDER BY, illetve mégis, de csak ha TOP is van, meg ilyenek. Megjegyezhetetlen.

Ha egy nézet több tábla összekapcsolásával alkotta meg azt a „táblát”, amit a felhasználó lát, a módosítás úgy változik, hogy lehet módosítani, de csak olyan UPDATE fut le, ami egyszerre egy időben csak egy háttértáblát módosít.

A nézetek biztonsági funkciójánál fontos kérdés, hogy a nézeten keresztül idegen adatot tudok-e macerálni. Mit tesz egy olyan módosítás esetén, amely láthatatlan, de létező adatra vonatkozik? Többé-kevésbé az elvárásoknak megfelelően viselkedik, bár sarokba lehet szorítani olyan UPDATE-tel, ami egy látható sort láthatatlanná módosít, mert megteszi. Ha valakinek ez a viselkedés nem tetszik, a nézet létrehozási parancsába a végére vegye fel a WITH CHECK OPTION-t, így:

```
ALTER VIEW KisCustomers AS
SELECT CustomerID, LastName, FirstName
FROM Customers
WITH CHECK OPTION
```

Egy másik opció pedig lehetővé teszi, hogy a nézet definícióját soha többé senki ne tudja elérni (beleértve magunkat is), melyhez a WITH ENCRYPTION kiegészítés alkalmazását javallom. Titkosítás után ne csodálkozzunk, ha a nézet többé nem szerkeszthető, nem scriptelhető, nem mozgatható, és összességében semmit nem lehet vele csinálni, ami a definíciójával kapcsolatos.

Végezetül emlékezzünk meg a VIEW-k teljesítménynövelő hatásáról, vagy más néven a materializációról! Az eddigi nézeteink tulajdonképpen nem álltak másból, mint egy elmentett SELECT-utasításból. Valahányszor a nézetet babráljuk, az alatta lévő táblán és táblákon dolgozunk. Ha egy bonyolult számítás van a nézetben, az bizony minden futtatáskor újra és újra kiszámítódik. Van azonban egy lehetőség, ami ezt a sok újraszámítást kiküszöböli, ez pedig a materializáció, vagy más néven az adattartalom lementése. Erre nincs külön parancs az SQL Serverben. A materializáció akkor következik be, amikor valaki custered indexet pakol a nézetre.

Sajnos a materializációnak 8-10 megjegyezhetetlen[14] feltétele van, úgyhogy aki ilyet csinál, majd a hibaüzenetek eligazítják, mit kell még tennie. Sok sikert!

[14] Persze, hogy tudom, de olyan száraz, hogy ha rendesen a könyvbe írnám, mindenki ültő helyében elaludna. Jó ez itt a láblécben. Kell bele WITH SCEMABINDING, COUNT_BIG() és egyedi értékű mező, nem lehet benne nem determinisztikus értékű mező, az indexnek pedig clusterednek kell lennie.

7 Tuning alapok I.

Most, hogy a lekérdezésekről már mindent tudunk *(kivéve a CTE-ket, az EXCEPT-et, INTERSECT-et, a HIERARCHYID-t, az XML-lekérdezéseket, a geokordinátákat és a pocedurális nyelvi elemeket – vö. mit adtak nekünk a rómaiak)*, rátérhetünk a teljesítményhangolás kérdésére. Azt már említettük, hogy a lekérdezések legjobb barátja az az index, amelyik hathatósan segít a lekérdezés adatainak előállításában. Arra is történt utalás, hogy ha egy lekérdezéshez nincs jó index, az SQL Server végig fogja kurkászni a táblát az adatok elérése érdekében. Mi mást tehetne?

Sőt, még az is igaz, hogy az SQL Server maga dönt a megfelelő indexek kiválasztásáról, mert ugyan ki látott már olyat, hogy egy lekérdezésben mi magunk megadjuk a használandó indexet? Én láttam. És hamarosan mindannyian látni fogunk ilyet. De ez csak kivétel lesz, ami erősíti a szabályt:

Az SQL Server maga választja ki a lekérdezéshez szükséges indexeket.

Mégpedig oly módon, hogy a lekérdezés adatait veszi szemügyre. Ha egy adathalmazban egy érték ritkán fordul elő, és van rá index, biztosan az indexet választja. Ha azonban a tábla fele egy adott értékkel van feltöltve (férfi-nő értékek), akkor sem fog indexet használni, ha van az adott mezőre index, mert olcsóbb számára a tábla felét minden cécó nélkül visszaadnia, mint hosszan indexszel bohóckodni.

Okos, ugye?

Azonban van itt egy tyúk-tojás probléma. Ahhoz, hogy ki tudja választani a megfelelő indexeket, tudnia kell, hogy egy adott adatból hány darabot fog visszakapni a lekérdezés **végén**. Amit a lekérdezés befejezéséig nem tudhat előre, ugye? Akkor mi alapján dönt?

7.1 Indexek, statisztika, szelektivitás. Használja? Nem használja?

Fontos fogalom a lekérdezés szelektivitása, ez határozza meg, vajon egy jónak tűnő indexet valóban érdemes-e használni, vagy olyan sok sort ad vissza a lekérdezés, hogy felesleges vesződni az indexszel. A szelektivitás pedig az úgynevezett indexstatisztikából derül ki.

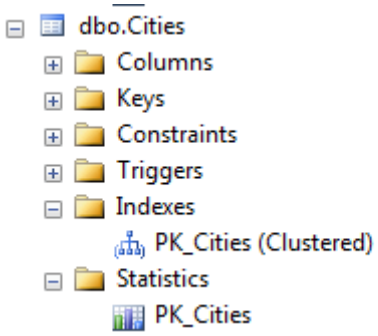
Amikor egy mezőre indexet készítünk (sőt, néha anélkül is), az SQL Server létrehoz egy statisztikának nevezett valamit, amit arra használ, hogy előre tudja, hány sort fog érinteni az adott lekérdezés. A statisztika alapján egy WHERE-feltételről előre tudja, hogy az 10, 100 vagy 1000 sort ad vissza. Már amennyiben a statisztika nem elavult. Ha nem bántalmazzuk az adatbázis beállításait, a statisztikák nem elavultak, mert automatikusan létrejönnek minden index mellé, és a karbantartásuk is automatikus, ha egy bizonyos mennyiségű indexkulcs megváltozott.

Furfangos kérdés: készítettünk-e már indexet a Bank adatbázis bármelyik tábláján, azon belül bármelyik mezőn?

Azé a hangszóró, aki azt válaszolta, hogy igen, készítettünk. Amikor ugyanis a kulcsmezőnket megjelöltük a PRIMARY KEY jelzéssel, a háttérben létrejött egy nem is akármilyen, hanem fürtözött (clustered, jó kis félrefordítás, bocsánat, nem én voltam…) egyedi (unique) index. *(Gyorsan egy kis*

fogalommagyarázat: A clustered gyakorlati jelentése: e szerint a mező szerint fizikailag is sorba rendezte a rekordokat. Unique: nem lehet az indexben két azonos érték.)

És ha már így esett, vele párhuzamosan létrejött egy hozzá tartozó indexstatisztika is. Így néznek ki egymás mellett:



32. ábra - index és a hozzá tartozó statisztika

Indexeket egyes mezőkre vagy mezőcsoportokra hozunk létre, egy táblán akár többet is. Egy index a szülinap mezőre, egy index a városnévre stb.

A clustered indexnél időzzünk el egy picit! Mit jelent az, hogy fizikailag sorba rakta a rekordokat? Egy hasonlattal világítanám meg a kérdést: a budapesti telefonkönyv lapjain ténylegesen (fizikailag) helyes sorrendben vannak az „áldozatok” – bocsánat, az előfizetők. A telefonkönyvön tehát van egy vezetéknév+keresztnév clustered (fürtözött?) index.

Ha ez a hasonlat érthető, a hangszóró párját az kapja (hogy meglegyen a sztereó!), aki megmondja, hogy hány clustered index szerepelhet egy táblán. Igen, pontosan ugyanannyi, ahányféle fizikai sorrendje lehet egy telefonkönyvnek. Egy, egyetlenegy.

7.2 Indextípusok működési módja

Ezzel elérkeztünk az indexek működéséhez és használhatóságához. A telefonkönyves hasonlat annyira jó, hogy nagyon örülnék, ha én találtam volna ki, de sajnos loptam valakitől, és már nem is emlékszem, kitől. Maradjunk tehát a telefonkönyvnél.

Hogyan lehet kikeresni a telefonkönyvből az összes Gipsz Jakabot? Mi sem egyszerűbb, belepörgetünk a G-ig, továbbgörgetünk a Gipsz-ig, majd Gipsz Jakabig, és onnantól sorban egymás után jönnek a Gipsz Jakabok, ugrálgatni nem is kell. Az utolsó Gipsz Jakab után biztosra vehetjük, hogy nem lesz több, megállhatunk.

Akkor most keressük ki az összes Jakabot! Ööö… Jaj. De hát mit vacillálunk, van egy vezetéknév+keresztnév indexünk, benne a nevek helyes sorrendben… vagyis hogy izé… Ezen indexen belül a Jakabok akárhol lehetnek! Vagyis ez az index nem alkalmas Jakab-keresésre. Akkor mi marad? Oldalanként végiglapozni a teljes telefonkönyvet, kiszedve belőle a Jakabokat. Hát igen. Az SQL Server esetén is így van. Ha egy index sorrendje rossz, akkor az rossz index, hiába digitális, és nem papíralapú.

A teljes tábla végignézése is egy lekérdezési stratégia az SQL Serverben, úgy hívják, Table Scan, illetve ha van a táblán clustered index (normál esetben mindig van), akkor Clustered Index Scant fogunk látni, de ez attól még ugyanaz.

A következő feladat a hangszóróért *(hű, ez már 3/5 surround!)* annak megválaszolása, hogy milyen lekérdezési stratégiával lehet a budapesti telefonkönyvből kikeresni az összes duguláselhárító kisiparost.

Ön nyert, Table Scan[15] az egyetlen létező megoldás. Melynek költségigényét ne feszegessük. Nem véletlen, hogy duguláselhárítót szaknévsorból keresünk. És ezzel elérkeztünk a további, nonclustered (értsd: hagyományos) indexekhez. Mi lenne, ha a szaknévsor nem tartalmazná a szaki elérhetőségét, csak egy mutatót a Hivatalos Telefonkönyv adott oldalán egy adott sorra? Akkor megvalósulna a hagyományos index, melynek használata a következő.

Ha egy adott szaki elérhetőségét keressük, gyorsan megtaláljuk a szakmát az indexben, majd hopp, előkapjuk a telefonkönyvet, felütjük a megadott oldalon, és már meg is van a kívánt telefonszám.

Ha arra vagyunk kíváncsiak, van-e a városban patkolókovács, felütjük a szaknévsort, amiben vagy megtaláljuk a patkolókovácsokat, vagy nem, de akár így, akár úgy, a keresés befejeződött, a telefonkönyvre nem kell átugrani, mert senki nem kérdezett telefonszámot.

Ha Sopronban játsszuk ugyanezt a játékot, és ki akarjuk keresni a fogorvosok telefonszámát, az egy másik helyzet, mert Sopronban minden második lakos fogorvos (az osztrákok miatt), így halál felesleges a szaknévsorból átugrani a telefonkönyvre meg vissza, következő fogorvos, meg vissza, oda, vissza, oda, vissza, ilyen esetben a szaknévsort pihenni hagyjuk, és végignyaljuk a telefonkönyvet. A lekérdezés szelektivitása kisebb annál, aminél megérné vesződni az indexszel.

Ezzel egy általános tanulsághoz jutottunk. Nem mi határozzuk meg, hogy egy index jó lesz-e egy lekérdezés gyorsítására, és még csak nem is az SQL Server, hanem az adat maga! Az adatok eloszlása, szelektivitása a kulcskérdés! Ha valamiből kevés van, jó lesz az index! Ha meg rengeteg, uccu neki, Table Scan. Nem érdemes erőltetni.

Ezzel át is vettük az indexelés elméleti alapjait, sajnálom, hogy csak három hangszórót osztottam ki, ez van, lépjünk túl ezen a hiányosságon! *(Van bal és jobb, meg center. Nem elég?)*

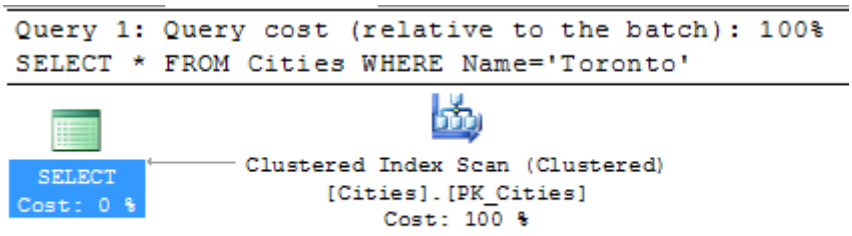
7.3 SQL-lekérdezések végrehajtási tervének összehasonlítása

Vegyük az alábbi lekérdezést, mely egy várost keres:

```
SELECT * FROM Cities WHERE Name='Toronto'
```

Ha lefuttatjuk, pillanatok alatt kiadja a semmit – merthogy nincs ilyen városunk a táblában. Gyors? Gyors. De hogy mennyire, arról a végrehajtási terv megtekintésével győződhetünk meg. Nyomj CTRL+L-t, vagy válaszd a Query menüből a Display Estimated ExecutionPlan menüpontot:

[15] A Scan szónál adódott volna egy találó magyarítás, a szánkázás, táblaszánkázás, de végül elvetettem, mert fájt leírni.



33. ábra - Clustered Index Scan lekérdezési stratégia

Technikai értelemben nem annyira gyors, mint lehetne. Merthogy egész egyszerűen végigtalpal a Cities táblán *(emlékezzünk, Clustered Index Scan=Table Scan, csak így írja ki a kis butus)*. Ez olyan kérdés volt, amire nincs index, de találat sem. Mint a hóhányó szakma jeles képviselői a telefonkönyvben.

Egyébként ha egy felhasználóként egy táblát kérdezgetünk, megdöbbentően gyors az SQL Server. Szinte mindegy, hogy hány rekordon, akár több millión futtatjuk, hasít. Pedig a nemlétezés megállapításához is végig kell másznia a táblán.

Egymillió város mindössze olyan 8 megabájtnyi adat, aminek az ezerszerese fér el egy pendrájvon, azaz egymilliárd város még simán hordozható kategóriájú adatmennyiséget jelent. Ez azt jelenti, hogy akárhogy küzdünk, a város táblánk egy az egyben be fog férni a memóriába, és másodpercenként több milliárd gépi kódú utasítást végrehajtó processzorunk is úgy falja fel, mint én reggelire a rántottát.

A tipikus fejlesztői tévedés, amikor a saját gépén minden hasít, és azt gondolja, ez majd éles helyzetben is így fog tekerni. Hát nem. Egyrészt a felhasználók száma garantáltan több lesz, mint egy. Másrészt ha egy ilyen lekérdezés nem óránként egyszer, hanem percenként tízezerszer fut le, ott már minden milliszekundum számít. Percenként tízezerszeri lefutást pedig még aprócska weboldalakkal, néhány száz látogatóval is el lehet érni, ha valaki idétlenül írja meg a kódját, és még nem is gyorsítótáraz (kessel). Tapasztalatból mondom! ☺

Az a tény, hogy az adatbázis-alkalmazások 90%-a index nélkül kezdi meg az életét, nem jelenti azt, hogy sokáig húzni is fogja így. Az adatmennyiség és a felhasználószám növekedésével garantáltan be fog borulni. Lássuk, miért!

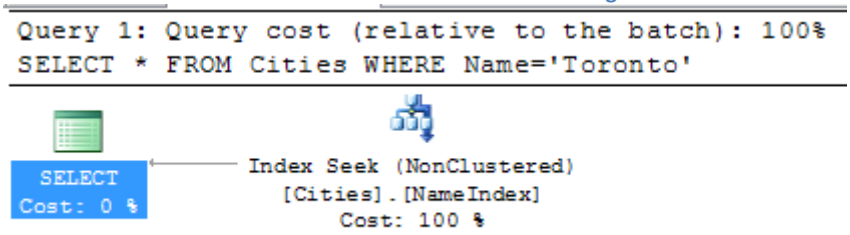
Csináljunk indexet a táblára, és nézzük meg, ennek hatására hogyan változik a feldolgozás!

```
CREATE INDEX NameIndex ON Cities(Name)
```

Lehetne összevissza bonyolítani ezt a parancsot, de az nem közelebb, hanem távolabb vinne a megértéstől, úgyhogy a hiperigényesektől azt kérem, hunyják be a szemüket. Igen, tudom, NONCLUSTERED lett (mi más?), és hát FILLFACTOR meg mifene – hagyjuk. Megcsinálja a Name mezőre az indexet? Meg. A negyedik hangszóró az enyém.

Az előző lekérdezést változatlanul le-nem-futtatva (CTRL-L) egy másik végrehajtási tervet kapunk:

34. ábra - index Seek lekérdezési stratégia



Első pillantásra talán ugyanolyannak látszik, mint az előző, de ez nem igaz. Egyrészt látjuk a feliratból, hogy a NameIndexet használja, másrészt az ikonja is más, átmegy rajta a kék villám, míg az előzőekben L-betű szerűen feküdt alatta. A megnevezése is más, ez nem Scan, hanem Seek, keresés, vagyis az ábráról összességében az olvasható le, hogy a NameIndexet használta, mégpedig helyesen, bejárva a benne megbúvó fát.

Kellene csinálni egy egymillió soros ellenpróbát, hogy mi a különbség a két stratégia között!

Ezzel ne fertőzzük meg a szép kis banki tábláinkat, készítsünk egy direkt erre a célra létrehozott táblát, mondjuk dátumokkal, így:

```
CREATE TABLE Osszevissza_datumok
(
ID INT PRIMARY KEY IDENTITY,
Datum DATETIMEOFFSET
)
```

Egymillió sort pedig azzal az ügyes trükkel fogunk belevinni, hogy „tudjuk”, hogy a GO utasításnak meg lehet adni, hogy az adott batchet hányszor futtassa le, így elég egyetlenegy hibátlan INSERT-et összetákolnunk, az egymillió már megy magától:

```
INSERT Osszevissza_datumok(Datum)
VALUES (DATEFROMPARTS(1900+RAND()*100, RAND()*11+1, RAND()*27+1 ))
GO 1000000
```

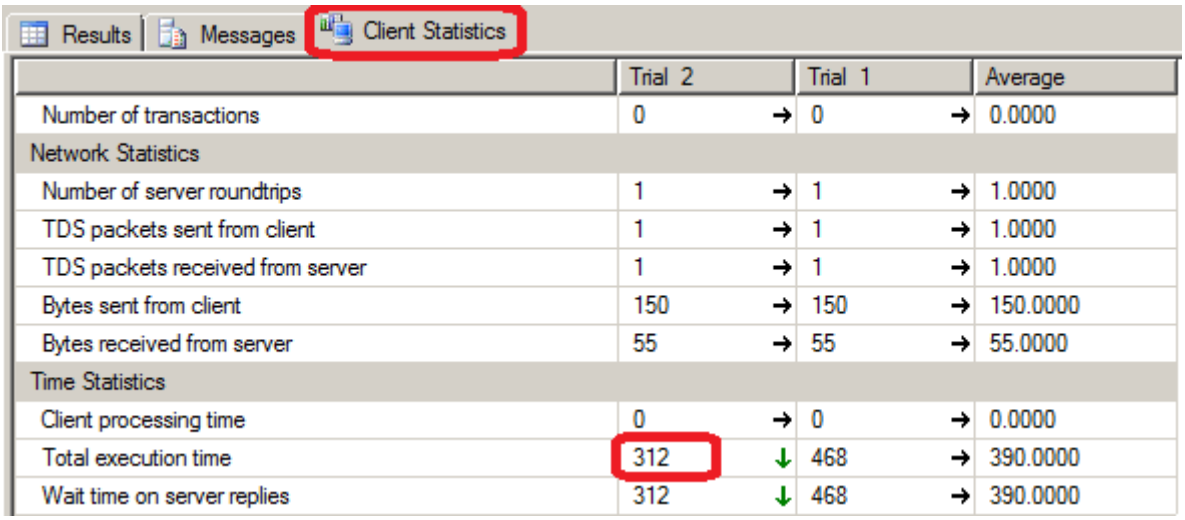
Akinek esetleg nem működik a DATEFROMPARTS() függvény, az most bukott le, hogy nem SQL 2012-t használ. A milliós GO korábban is működött már. *(Amíg az egymillió sor készül, egy szomszédos lekérdezőablakból utánaérdeklődhetünk, hol tart a folyamat: select count(*) from Osszevissza_datumok.)*

Egyébként létezik ennél sokkal gyorsabb sorgenerálás is, sőt, ez talán az egyik leglassabb módszer. Minden INSERT külön tranzakció… Gyenge. De legalább nem gépelős.

Ha kész az egymillió dátum, jöhet a lekérdezési teszt. Már az első futtatás előtt kapcsoljuk be az időmérő készüléket, amelyet a menüsoron találunk egy megnevezhetetlen ikon mögött, valamint a Query menüben, Include Client Statistics néven! Ezzel felvértezve pontosan látni fogjuk a különbséget az egyes futtatások között. És akkor lássuk az index nélküli, nem létező adatra vonatkozó keresést! Mint a fenti INSERT parancsból látszik, minden dátumunk 1900 utáni, tehát egy 1800-as évekbeli lekérdezés garantáltan a semmibe nyúl:

```
SELECT * FROM Osszevissza_datumok
WHERE Datum='1848.03.15'
```

A lekérdezési terv index nélkül – nem meglepően – egy Clustered Index Scan, teljes táblavégiggyaloglás. Szót sem érdemel. Ha most ténylegesen le is futtatjuk, a statisztika az én gépemen ezt hozta:



	Trial 2	Trial 1	Average
Number of transactions	0 →	0 →	0.0000
Network Statistics			
Number of server roundtrips	1 →	1 →	1.0000
TDS packets sent from client	1 →	1 →	1.0000
TDS packets received from server	1 →	1 →	1.0000
Bytes sent from client	150 →	150 →	150.0000
Bytes received from server	55 →	55 →	55.0000
Time Statistics			
Client processing time	0 →	0 →	0.0000
Total execution time	312 ↓	468 →	390.0000
Wait time on server replies	312 ↓	468 →	390.0000

35. ábra - nem létező rekord keresése egymillió sor között

300 milliszekundum egymillió soron. Hát emiatt kár lenne vergődni az indexekkel, mondhatná valaki. De ne feledjük el, hogy egy felhasználó dolgozik „párhuzamosan”, és a teljes tábla vélhetőleg bent van a fizikai memóriában. A kérdés inkább az, hogy objektíve hányszorosára gyorsul ez a lekérdezés, ha odaadjuk neki a megfelelő indexet.

A kliensoldali statisztika jó dolog, de a szerveroldali még jobb. Kapcsoljuk be a 8k-s lapok mozgását lemérő statisztikát ezzel a paranccsal:

```
SET STATISTICS IO ON
```

Ezt követően lefuttatva (és ténylegesen lefuttatva, F5!) a fenti parancsot, a Messages fülön a következő érdekes sorokat olvashatjuk:

```
Table 'Osszevissza_datumok'. Scan count 1, logical reads 2853, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.
```

Magyarra fordítva: a táblánkhoz egyszer kellett hozzányúlnia, amikor is 2853 darab 8k-s lapot olvasott végig, ami mind a memóriában volt (logical read= memóriaolvasás), és nulla darabot kellett a merevlemezről beolvasnia – merthogy mind a memóriában volt. *(A teljes adatmennyiségünk tehát 2853 * 8k = 22 MB, egymillió dátum plusz egymillió integer.)*

Most tegyünk indexet a Datum mezőre, és futtassuk újra!

```
CREATE INDEX DatumIndex ON Osszevissza_datumok(Datum)
```

Az előző statisztika helyett most ezeket a számokat kapjuk:

```
Scan count 1, logical reads 3
```

Hohó! A gyorsulás több mint 900-szoros! Kilencszázszoros! Halljátok? Ki-lenc-száz-szo-ros!

Ez olyan döbbenetes gyorsulás, ami megmagyarázza, miért marad életben az a rengeteg idióta alkalmazás, amelyik másodpercenként tízezer egyforma kérdéssel bombázza az SQL Servert. Csak azért, mert egy jó indexszel párezerszeres gyorsulást lehet elérni, és ez menti meg a helyzetet.

7.4 Indexbeszögelő optimizer hintek

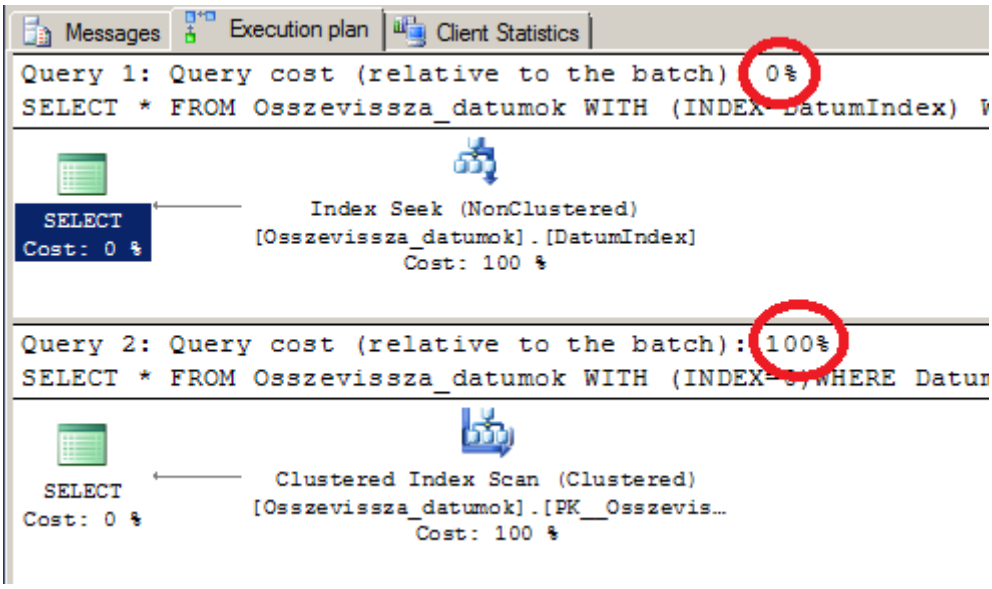
Ha már tuningolunk, emlékezzünk meg az úgynevezett optimizer hintekről, melyeket élő (produkciós) környezetben nem használunk, hiszen azt feltételezzük, az SQL Server úgyis okosabb nálunk, ami igaz is. Előfordulhatnak azonban olyan esetek, amikor látszólag nem annyira okos, és jó lenne tárgyilagosan összehasonlítani, hogy az általa választott megoldás tényleg annyival jobb-e, mint ami első látásra logikusnak tűnik.

Rengeteg optimizer hint létezik, mi most csak egyet használunk, azt, amelyikkel indexhasználatot (vagy nemhasználatot) tudunk kikényszeríteni.

Az előző lekérdezést másoljuk le egymás alá két példányban, és az egyikre varrjuk rá az indexet, akár tetszik neki, akár nem (egyébként tetszik, hisz ezt használta magától is) a WITH (INDEX=DatumIndex) utasítással, míg a másikról tiltsuk le az indexet a WITH (INDEX=0) utasítással, így:

```
SELECT * FROM Osszevissza_datumok WITH (INDEX=DatumIndex) WHERE
Datum='1848.03.15'
SELECT * FROM Osszevissza_datumok WITH (INDEX=0) WHERE
Datum='1848.03.15'
```

Majd a két sort együtt kijelölve nyomjunk egy nagy CTRL+L-t (Estimated Execution Plan), és vizsgáljuk meg a két végrehajtási tervet! Ami érdekes lesz, az a két lekérdezés végrehajtási teljesítményének egymáshoz való aránya, melyet pirossal bekarikáztam:



36. ábra - két lekérdezés teljesítményének összehasonlítása

Megállapíthatjuk, hogy a tábla végigolvasása vitte el az egyesített végrehajtásból az idő[16] 100%-át, majd a fennmaradó 0%-nyi idő alatt az indexes lekérdezés is lefutott. ☺ Tehát a közel ezerszeres gyorsulás nem vicc, itt is látszik, hogy töredékidő alatt fut le az indexszel gyorsított lekérdezés.

7.5 Lábas volt a fenőnevem

Egy másik tuningolási érdekesség a csillag karakter használata, illetve nemhasználata. Sokszor szoktam mondani, hogy a csillag „tiltott” karakter, most eljött az ideje, hogy megtudjuk, miért. Költözzünk vissza a Customers táblára, és kérdezzük le Damon Hillt:

```
SELECT * FROM Customers WHERE LastName='Hill'
```

Illetve akár ne is kérdezzük le, az eredmény úgysem érdekel, hanem rögtön nézzük a végrehajtási tervet (CTRL+L)! Vagy még jobb, ha a fenti okfejtés ismeretében megpróbáljuk kitalálni, mi is lehet az. Végül is van még hangszóróm, odaadhatom.

Aki arra tippelt, hogy Clustered Index Scan lesz a „nagy terv”, nem tévedett, övé a hangszóró. Index nélkül keressük Hillt, mi mást tehetnénk, mint egy komplett táblaolvasást? Ez bemelegítőkérdés volt, hadd ne tegyem ide a képernyőképet, csak ismételném korábbi önmagamat.

Az előzőek szerint dobjunk fel egy indexet a LastName mezőre, és ismét vizsgáljuk meg a „haditervet”!

```
CREATE INDEX LastNameIndex ON Customers(LastName)
```

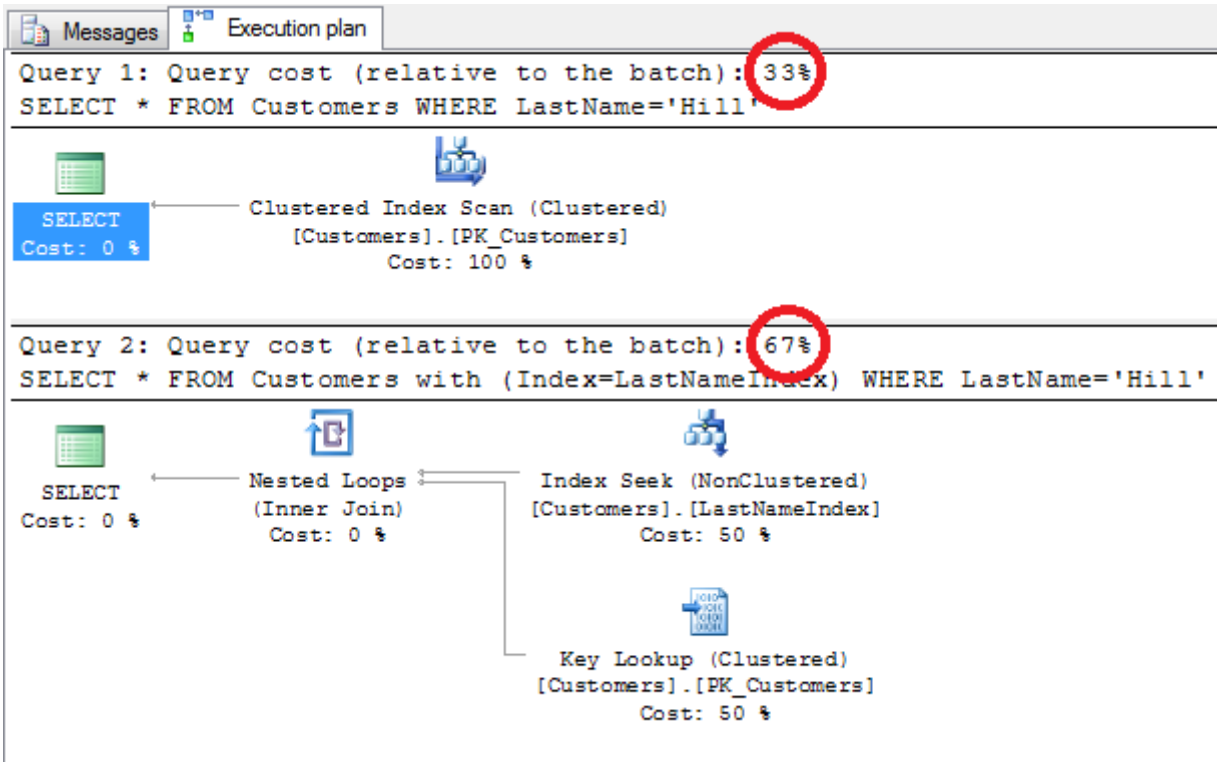
Az index frankón létrejött – ám az SQL Server nem használja! Akinek az jött le a végrehajtási tervből, hogy ezt használja, adja vissza a hangszóróját!

[16] Ez az „idő” valójában nem idő, hanem IO-művelet, a lekérdezéshez felhasználandó 8k-s lapok száma. Mivel a lekérdezés idejét a legnagyobb mértékben az határozza meg, hány 8k-s lapot kell megmozgatni, én a továbbiakban időt írok, mert ettől még a mondat is olvashatóbb lesz. Fejben mindenki fordítsa át az „idő”-t „8k-s lap”-ra! Köszönöm.

Szerencsére kezünkben van az eszköz, amivel ki tudjuk kényszeríteni az indexhasználatot, és ki tudjuk mérni, hogy vajon miért nem használja? Futtassuk az alábbi két parancsot együtt kijelölve, és mindjárt kiderül a turpisság:

```
SELECT * FROM Customers WHERE LastName='Hill'
SELECT * FROM Customers WITH (INDEX=LastNameIndex) WHERE
LastName='Hill'
```

Az első az eredeti lekérdezés, a második pedig az a változat, amelyikben ráerőszakolom azt az indexet, amelyiket valamilyen okból nem hajlandó használni. A végrehajtási tervek a következőképpen alakulnak:



37. ábra - Mitől ilyen bonyolult az indexes lekérdezés?

A piros karikák alatt látszik, hogy az indexmentes lekérdezés az összesített idő 33%-át, az indexes pedig 67%-át vitte el, tehát az indexes lekérdezés ebben az esetben kétszer annyi időt emészt fel, mint a sima Table Scan. Hmmm. Biztosan az lehet a háttérben, hogy a második végrehajtási terv valami hiperbonyolult, értelmetlen akármi lett, amiben még JOIN is van. Aki erre tippelt, nem jár messze az igazságtól, a kérdés már csak az, mi ez a káosz ottan? Ha ez a csúnya komplexitás nem csúszik be, biztosan az indexes lett volna az olcsóbb. De becsúszott.

Most jön a vaslogika.

Milyen adatok szerepelnek a LastNameIndex indexünkben? Benne vannak a vezetéknevek, szépen bináris fába felfűzve, könnyen kereshetően. Emellett benne van az elsődleges kulcs mint mutató, amivel arra a sorra mutat, amelyikből származik.

És mi melyik mezőket kívánjuk kilistázni a fenti lekérdezéssel? A mindenséget, a csillagos eget.

Emiatt ha a delikvensünket villámgyorsan megkeressük az indexszel, még csak két adathoz jutottunk hozzá, a vezetéknevéhez és az elsődleges kulcshoz. Ez a kívánt adatmennyiség fele se. A kulcsérték birtokában most le kell fúrni a tábláig, Damon Hill megfelelő soráig, és fel kell olvasni a maradék adatokat. Ez lenne a Key Lookup lépés, mely a kulcs alapján a többi adatot felhozza a mélyből. Most akkor van két adathalmazunk, az, ami az indexből jött, és az, ami a táblából. Ezt a kettőt a kulcsmező segítségével összedzsoinoljuk (Nested Loop), és már készen is vagyunk.

Érdemes az egérrel a Nested Loop felé tartó vonalakra állni, mert megmutatja, melyik nyílon hány sor és hány bájt jön. Bizony. Az indexből jön 22 bájt, és további 66 bájt a táblából.

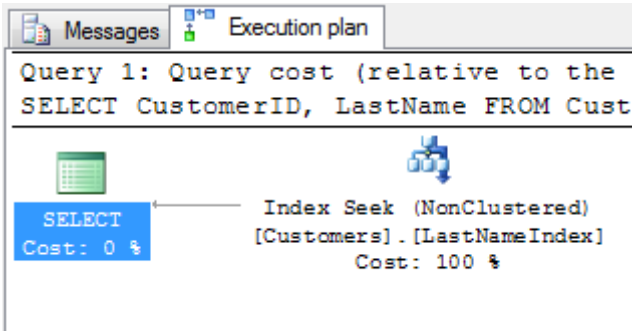
Egy dolgot azért tisztázzunk: ebben a példában a teljesítményeltérés abból adódik a Table Scan javára, hogy van vagy öt vevőnk, tehát a Table Scan biztosan olcsóbb bárminél, mert 1 darab 8k-s lapot kell megmozgatni. A Fejlesztő Tévedése szindrómát láthatjuk teljes pompájában virítani. Minek ide index? Ja, hogy élesben majd nem öt, hanem ötvenezer kuncsaft lesz? Ki ér rá ezzel szembesülni? Haladjon a fejlesztés, haladjon!

Ha tehát a Customers táblát megtolnánk pár ezer vevővel, hamar fordulna a kocka, és az indexelés-Key Lookupolás a maga 3 darab 8k-s lapjával kenterbe verné a Clustered Index Scant.

De még e pici tábla esetén is menthető a helyzet. Rossz ötlet lenne az indexet rászögelni a lekérdezésre, azt már láttuk. De mi lenne, ha engednénk egy picit az igényeinkből, és nem minden oszlopot kérdeznénk le, hanem mondjuk csak az ID-t és a vezetéknevet? Hagyjuk a csillagot békén, annak az égen a helye, nem a lekérdezésünkben. Erre gondolok:

```
SELECT CustomerID, LastName FROM Customers WHERE LastName='Hill'
```

A lekérdezés végrehajtási terve pedig a következő lett:



38. ábra - egy szép Index Seek

Hohó! Elkezdte használni az indexet, sőt, a Lookup és a Nested Loop is eltűnt!

A jelenség oka az, hogy csak olyan adatot kérdeztünk le, ami már magában az indexben is megtalálható, tehát nincs értelme lefúrni a táblába. Az ilyen helyzeteket fedő (covering) lekérdezésnek, fedő indexnek hívjuk, és sokkal többet ér annál, mint amit nyilvánvalóan már most is hozott.

A fedő indexes lekérdezés ugyanis azáltal, hogy nem megy le a táblához, garantáltan nem akadályozza a többi felhasználót, különös tekintettel az adatmódosításokra. Garantáltan nem akadályozza az adatmódosításokat, hiszen ott sincs. Egy ügyes fedő indexszel megintcsak

tizenezerszeresére lehet növelni egy alkalmazás *(látszólagos)* teljesítményét, pedig ebben az esetben csupán a várakozásokat számoljuk fel, ha ügyesek vagyunk.

Nem véletlen, hogy gyakran készítünk többmezős indexeket, hogy minél fedőbbek legyenek, illetve nem véletlen, hogy az indexekbe behelyezhetők úgynevezett vendégoszlopok az INCLUDE kulcsszó használatával, amelyek a sorba rendezésben ugyan nem vesznek részt, de legalább fednek. Ha lemondunk a csillagról. A halálcsillagról.

39. ábra - halálcsillag

A fedés egyébként még olyan esetekben is helyzetbe hoz indexeket, ha a sorrendjük ugyan rossz, de tartalmilag rendben vannak, mert ilyenkor az SQL Server előszeretettel csinál „Table Scan”-t magán az indexen, minitáblának tekintve azt. És igaza van. Még egy a lekérdezés szempontjából helytelen sorrendű index szánkázása is gazdaságosabb magának a táblának a végigtekerésénél, mivel egyrészt kisebb adathalmaz, másrészt ismét befigyel a másokat-nem-akadályozunk jelenség.

7.6 Balról zárj! A LIKE utasítás

Kezdő tuningismereti áttekintésünkben végezetül a balról zártság fogalmával ismerkedünk meg. Azt már láthattuk, hogy egy összetett index (vezetéknév, keresztnév, telefonkönyv) második tagja a teljes adathalmazra vetítve „kaotikus” sorrendben van, avagy hol is van Jakab a telefonkönyvben?

Ugyanez a káosz kicsiben is megfigyelhető az egyes szöveges indexek karakterein. Értelmesebben kifejtve: a vezetéknév index legelső karaktere ábécérendben van. A második karaktere pedig az elsőhöz képesti ismétlődő kisábécékben. A harmadik karakter a másodikhoz képest még kisebb ábécék garmadája. Ha azonban a bevezető karakterek nélkül, összefüggéséből kiragadva nézzük például a harmadik karaktert, mini ábécék ikszezerszer ismétlődve, tehát gyakorlatilag kaotikus sorrendben van. Avagy keresd ki a telefonkönyvből az összes embert, akinek a vezetéknevében a harmadik betű „z”!

Az ilyen lekérdezések végrehajtásában az indexek csak annyiban tudnak segíteni, hogy ha sorrendileg nem is, de fedőként részt tudnak venni a keresés gyorsításában.

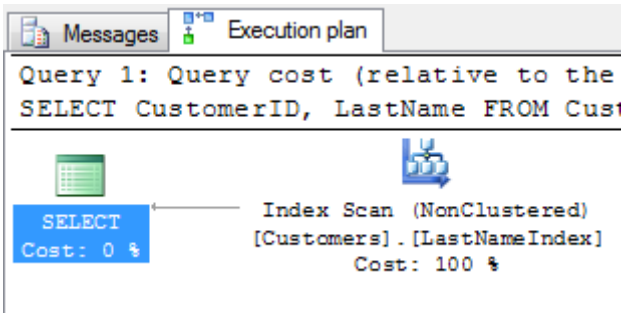
A LIKE utasítás segítségével szoktuk leggyakrabban elkövetni a balról nem zárt lekérdezéseket, és noha ez nem az egyetlen jó módszer az indexekkel való kitolásra, de gyakoriságban ez viszi a prímet.

Másik közismert módszer szöveges mezőkben *(például terméknevek, könyvcímek)* bizonyos szavakra keresni, mert ott is az a helyzet, hogy az első szó még balról zárt kifejezésnek számít, de a második már nem.

Fentebb már láthattuk, hogyan néz ki egy szabályos indexhasználat, ezt itt nem jelenítem meg ismét. *(Emlékeztetőül: Index Seek, Kék Villám.)* Most keressük ki azokat a vevőinket, akiknek i betű van a nevében:

```
SELECT CustomerID, LastName FROM Customers WHERE LastName LIKE '%i%'
```

Az ehhez tartozó végrehajtási terv pedig így néz ki:



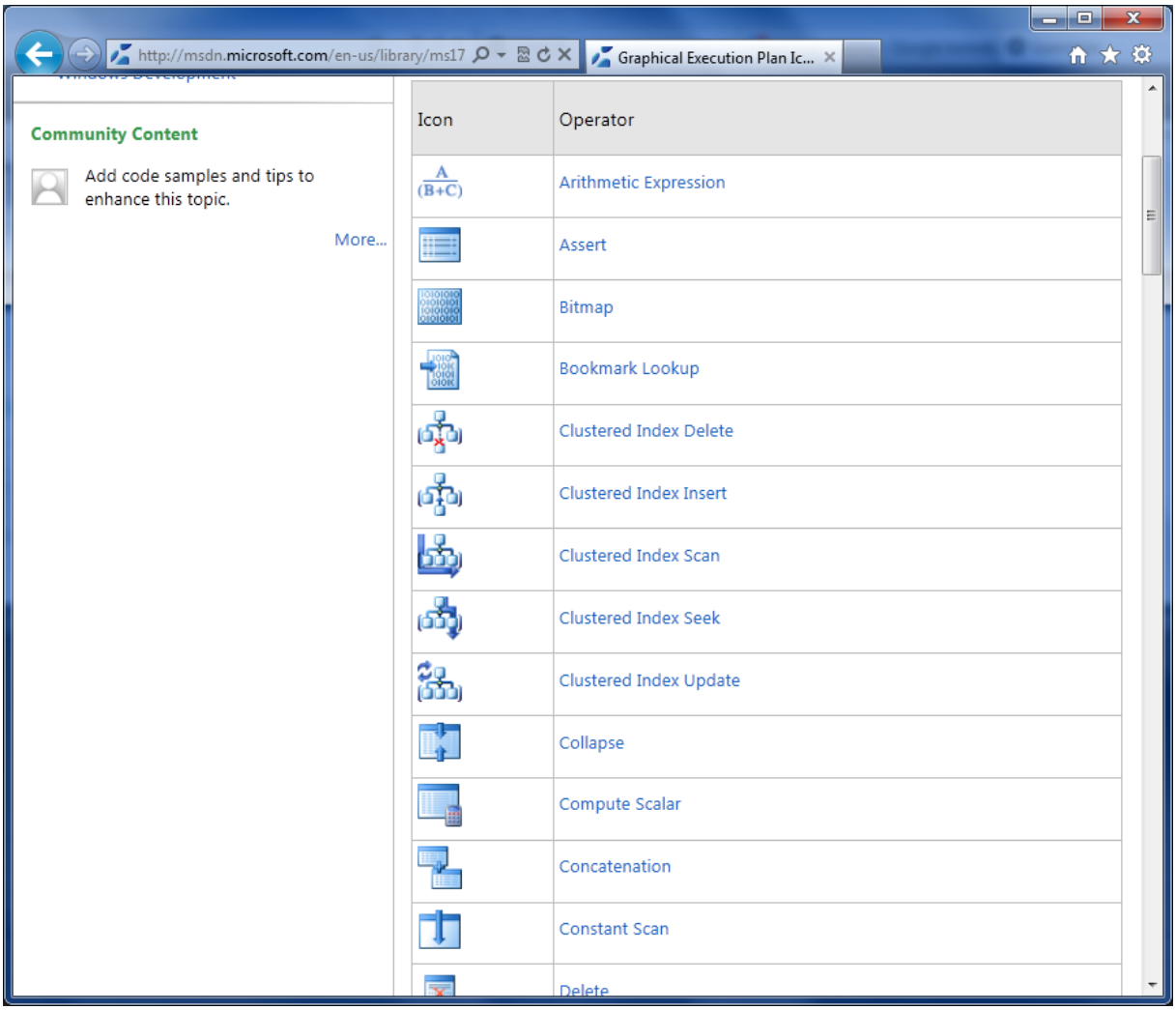
40. ábra - egy csodálatos Index Scan

Amit látunk, az maga a fedés. Felhasználja tehát a LastNameIndexet, de annak a sorrendje a jelen lekérdezés vonatkozásában hulladék, viszont legalább lefedi a kérdéses lekérdezést. A stratégia nem Seek, hanem Scan, kék villám nyilacska helyett kék L-betű nyilacska van. Az történik, hogy az indexünket mint minitáblát szánkázza végig, mert olcsóbb ezen szánkózni, mint a tényleges táblán. But that's it, ahogy a művelt orosz mondaná.

Mindebből az következik, hogy próbáljuk elkerülni a balról nyitott lekérdezéseket? Hát, ez nem következhet, mivel ha egy alkalmazásnak erre van szüksége, ezt fogja csinálni. A tanulság inkább az, hogy ilyen esetben az index hatékonysága töredéke annak, mint amire számítunk – de még így is megéri leindexelni ezt a mezőt, mert a fedés az mégiscsak fedés.

Akit pedig nem vert meg a magyar nyelvvel a sors, megpróbálkozat a Full Text Index bevetésével, ami ugyan a karakterenkénti balról nyitottság ellen nem véd, de ha egy szöveges mezőben keresünk egy szót, az is balról nyitott kifejezés, az ellen például véd. Milyen kár, hogy kis hazánk egy fehér folt és egy fekete lyuk együtt a Full Text Index térképén. Ezért ez utóbbival most nem is foglalkozunk, majd talán 2016-ban.

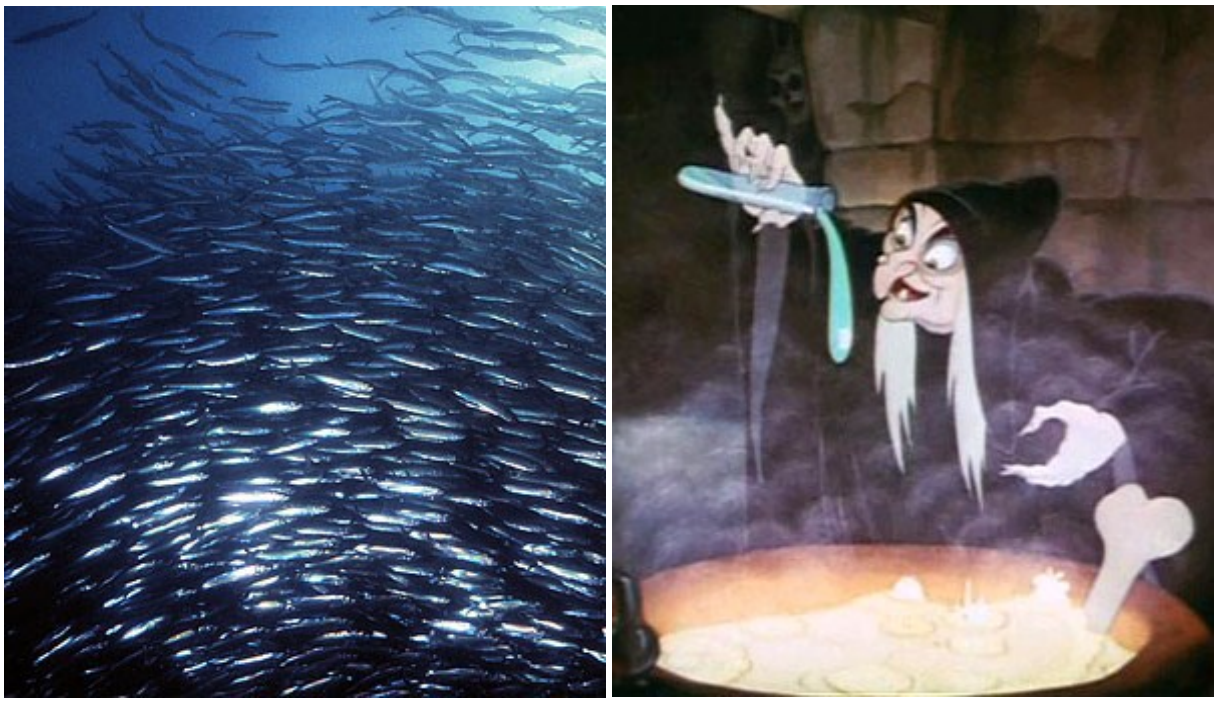
Ezzel a tuninglecknénk első része véget ért, visszatérünk még a témára a tranzakcióknál. Aki esetleg úgy gondolja, most aztán jelentős mélységekbe kalauzoltam el, hadd lombozzam le: a felszínt karcolgattuk. Nem mondom, hogy ezzel a három lekérdezési stratégiával nem lehet okosakat mondani egy lekérdezésről, de mi van a maradék 40 ikonkával? Ízelítő a Books Online idevágó fejezetéből, különös tekintettel a jobb oldali gördítősáv elkeserítő méretére:



41. ábra - végrehajtási tervelemek garmadája

8 Data Manipulation Language

Mindezidáig egy-két szükséges példa kivételével elkerültem az adatmódosítási utasítások körét. Ezt azért tettem, mert az adatmódosítás és –felvitel egy másik világ. Míg a SELECT lubickol a többfelhasználós végrehajtásban, és a különböző felhasználók lekérdezései úgy száguldoznak egymás mellett, mint a heringek az örvényben, az adatmódosító utasítások gonosz boszorkák. Mindent maguk akarnak csinálni a saját kis barlangjukban, lehetőleg úgy, hogy a piszkos ténykedésükbe senki ne lásson bele, az átváltoztatás trükkje csak az övék, a tiéd meg az eredmény, nesze.



42. ábra - a lekérdezések (balra) és a módosítások (jobbra) diagramja

Ennek természetesen megvan a maga oka. Igen bután nézne ki, ha egy sort többen egyszerre módosíthatnának, mert mivé lesz a világ, ha egy terméket egyszerre, egyidőben eladunk A-nak is és B-nek is? Nem-nem! Amíg az A művelet véget nem ért, addig abba ne kontárkodjon bele senki!

Ezt a fajta szétválasztást az SQL Server zárolási rendszere oldja meg, teljesen automatikusan. Mi csak önfeledten módosítgatunk, ő meg bezár, kinyit, megfog és elenged. A zárolási rendszer ismerete az adatmódosításokkal kapcsolatos párhuzamossági és egyéb problémák elhárításának fontos eszköze.

Aztán itt van a módosítások mellékhatása, a módosítások módosítása. A korábbi fejezetekben lazán szórtuk az indexeket, és nem törődtünk azzal, hogy ez a későbbiekben milyen hátrányokkal jár esetleg. Merthogy a módosítások bizony az indexeket is módosítják. Ha egy adatot leindexelek, ezzel megduplázom, hisz belekerül az érték az indexbe is, akkor ennek minden módosítása dupla módosítást igényel, hisz az nem engedhető meg, hogy a rekordban tárolt értéket módosítom, az őt elérhetővé tévő indexet viszont nem.

Ez az okfejtés azt sugallja, hogy ha egy adatbázisban túlnyomórészt módosítások zajlanak, akkor azt ne indexeljük. Csudát! Ez a „gondolatmenet” a programozó alaptévedése című tévedésgyűjtemény szép kiegészítője. Minden UPDATE és DELETE mindaddig SELECT, amíg meg nem találjuk a

módosítandó sorokat, de utána átcsap valami egészen másba. A boszorkány hering hátán érkezik, ha szabad egy költői képzavarral élnem. Tehát az okos mondás inkább az, hogy ésszel indexeljünk ilyen esetben. Csodák nincsenek. Ha egy módosító utasítás WHERE-feltételében szerepel egy mező, ami alapján meg kell találni a sort, de erre nincs index, akkor vajon hogyan fogja megtalálni a célrekordot az SQL Server? Table Scan=Clustered Index Scan.

Ennyi elmélkedés után vegyük sorra az adatmódosító utasításokat!

8.1 INSERT utasítás

Az INSERT utasítással viszonylag gyorsan fogunk végezni, mivel egyrészt már korábban kénytelenek voltunk használni, másrészt meg annál sokkal többet nem is tud, mint amire már használtuk. Van egysoros INSERT, többsoros INSERT, vannak egy táblában írható és *(általunk)* írhatatlan mezők. Nem is a szintaxis itt a lényeg, hanem az INSERT használata éles helyzetben. Mert van itt egy bökkenő.

Alapesetben az INSERT után zárójelben felsoroljuk azokat a mezőket, amelyeknek értéket kívánunk adni akár a VALUES kulcsszó után, akár egy SELECT vagy egy tárolt eljárás eredményhalmazából. Mely mező(ke)t nem soroljuk fel? Hát például az IDENTITY, vagyis automatikusan növekvő mezőket nem mi írjuk, hanem a rendszer. Ezért adtunk hozzá új várost a Citieshez így:

```
INSERT Cities(Name) VALUES ('Piripócs')
```

Egynél több értéket is meg lehet adni a VALUES után, a bezárójelezett mezőértékek vesszős felsorolásával.

Illetve SELECT eredményhalmazát így:

```
INSERT Cities(Name) SELECT 'Kisujjszállás'
```

Ahol természetesen tetszőleges SELECT állhat, amelyik ugyanannyi és ugyanolyan típusú mezőt ad vissza, mint amit az INSERT vár. Vannak ennél bonyolultabb SELECT-ek, ezt az olvasók fantáziájára bízom.

Ha valaki mindenáron minden mezőnevet fel akar sorolni (miért tenné?), akkor élhet azzal a trükkel, hogy az öntöltődős mezők öntöltést kapnak, ha ezt az „értéket” adjuk be nekik: DEFAULT. Esetleg ha valaki szándékosan üresre akar „tölteni” egy mezőt, nem árt, ha tudja, hogy azokba a mezőkbe NULL-t kell beírni értékként. Logikus.

És most jön a bökkenő. Mi lett az ID mező értéke, amit nem mi töltöttünk ki? Igen gyakori helyzet, hogy az alkalmazásunknak szüksége lenne a szerver által generált ID-re, mert az aparekord alá gyermekeket szeretne nemzeni, de ezt sajnos most nem kaptuk meg. Milyen lehetőségeink vannak?

1. Ha mi írjuk az azonosítót, akkor tudjuk, mi az azonosító.
2. Valahogy le kellene kérdezni a most kiosztott azonosítót.
3. Az INSERT lehetne okosabb, és visszaadhatná önként az azonosítót.

Jó hírem van, mind a három eljárás működik. Mivel ezek közül a legelső a legkacifántosabb, kezdjük inkább a másodikkal! Az éppen kiosztott IDENTITY lekérdezésére többféle lehetőség is van, ezek

közül csak azt mutatom, amelyik a legkevesebb buktatót tartalmazza. Ez pedig a SCOPE_IDENTITY() függvény, amit így használhatunk közvetlenül az INSERT után:

```
SELECT SCOPE_IDENTITY()
```

Ez az utasítás pontosan az általunk legutoljára futtatott INSERT során kiadott IDENTITY értékét adja vissza. A tudálékosabbak kedvéért megjegyzem, hogy tudom, hogy van @@IDENTITY, meg IDENT_CURRENT(), de ezekkel a kezdők mindig felborulnak, úgyhogy hagyjuk, nincsenek.

A harmadik lehetőség elsőre egy picit jobb ötletnek tűnik, mármint hogy az INSERT adja vissza az ID-t, mert akkor nem áll fenn az a veszély, hogy a beszúrás és az azonosító lekérdezése közé beékelődik valami más utasítás. Ezzel viszont az a baj, hogy olyan kliensalkalmazás is kell hozzá, amelyik eltűri, hogy az INSERT visszabeszél. Merthogy alapból néma. Így tudjuk őt szóra bírni:

```
INSERT Cities(Name) OUTPUT INSERTED.CityID VALUES ('Piripócs')
```

Az OUTPUT kulcsszó mindhárom adatmódosító utasításnál használható, és azt tudja, hogy kihajigálja a kimenetre egy tábla formájában a még memóriában lévő, véglegesítés előtti rekordállapotot. Olyan, mintha az INSERT egyben SELECT is lenne. Merthogy mostantól az is.

Látunk a fenti utasításban valami nemlétező táblát, melynek neve INSERTED, ez az a memóriatábla, melynek tartalma egy mikroszekundum múlva a lemezre kerül, ebben vannak az újdonságok, immár teljesen csőre töltve, IDENTITY kiszámolva, alapértelmezett értékek kitöltve stb.

Már négy hangszórót kiosztottam, a surroundhoz viszont kell egy ötödik. Azt annak adom, aki kitalálja, hogy ha törlést végzünk, vajon mi lehet a törléskor elérhető memóriatábla neve, ha az INSERT-nél INSERTED, és a törlési utasítás a DELETE?

Ön nyert, DELETED a tábla neve. Gratulálunk az öt hangszóróhoz, bár ha jól emlékszem, a negyediket magamnak osztottam ki, de valójában már rég elveszítettem a fonalat hangszóróügyben, úgyhogy összességében fogalmam sincs.

De nem is a hangszóró a lényeges, bár aki kapott belőle ötöt, annak bizony az. Hanem hogy vajon hogyan hívják a memóriatáblát UPDATE művelet esetén?

Na vissza azokkal a hangszórókkal! Nem UPDATED! Nem!

De hát honnan is lehetett volna előre tudni? Az UPDATE ugyanis egy DELETE és egy INSERT egymás után. Így aztán UPDATE esetén van INSERTED is és DELETED is egyszerre. Egyikben a rekordok előző állapota, a másikban pedig a fényes jövő.

A harmadik lehetőség pedig, hogy mi magunk írjuk az azonosító értékét. Itt azonban ellentmondásba futottunk, mert ha mi írjuk, akkor minek az IDENTITY? A kérdés jó, és az a válasz, hogy itt nem elsősorban az IDENTITY felülbírálására kell gondolni (bár azt is meg lehet tenni a SET IDENTITY_INSERT ON utasítás kiadását követően), hanem egy másik fajta egyedi azonosító képzésre, az úgynevezett szekvenciára.

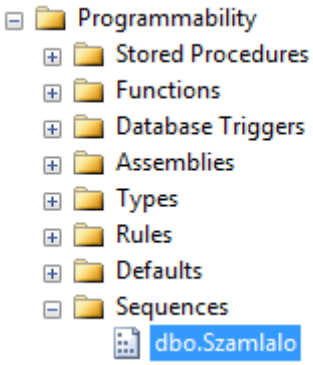
8.2 A szekvenciaobjekum

Az SQL 2012-ben jelent meg újdonságként a szekvencia, ami nem más, mint egy olyan számláló, amit nem a tábla kezel (ellentétben az IDENTITY-vel), hanem én, a programozó. Persze eddig is lehetett ilyesmit házilag összetákolni, és sokan éltek is ezzel a lehetőséggel, a szekvencia azonban sok tekintetben túltesz a házilag barkácsolt megoldásokon. Egyrészt beépítetten kezeli a többfelhasználós *(zárolási)* helyzeteket, másrészt gyorsítótáraz, harmadrészt lehetővé teszi, hogy az alkalmazások nagy blokkokban kérjenek maguknak azonosítókat, amiket aztán szabadon és az összeütközés veszélye nélkül használhatnak fel a későbbiekben. Ez az eljárás a vonalkódok kiosztásához hasonló, ott is minden felhasználó teljesen autonóm módon gazdálkodik egy előre kiosztott készletből.

Hozzunk létre egy szekvenciaobjektumot:

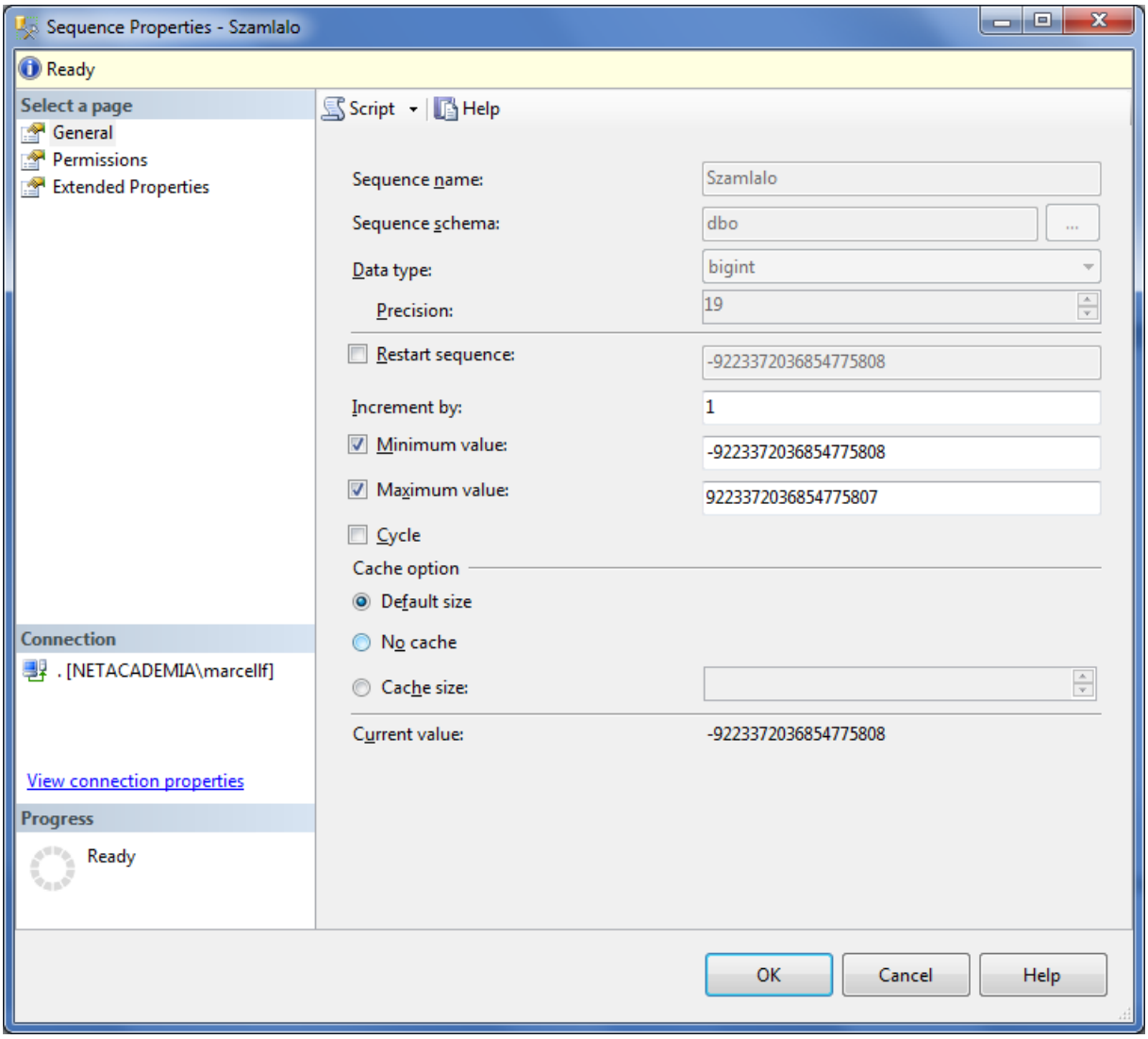
```
CREATE SEQUENCE Szamlalo
```

Ez a jószág bekerül a programozhatósági eszközök közé a fában, itt:



43. ábra - szekvenciaobjektum

Ha megnyitjuk a tulajdonságlapját, láthatjuk, hogy ez egy mi ez, és mi mindent állíthattunk volna be rajta létrehozáskor.



44. ábra - a szekvencia tulajdonságlapja

Egyrészt be tudtuk volna állítani az adattípusát mondjuk INT-re, merthogy most BIGINT. Másrészt beállíthattunk volna neki egy kezdeti értéket, mert a mínusz végtelenből jön majd fel. Ami egyébként nem baj, hisz hatszáz oldallal korábban megegyeztünk, hogy az a jó azonosító, amelyiknek nincs saját értelme, akkor viszont miért ne lenne jó a -9223372036854775808? Be lehet állítani a növekményét, a tól–ig határokat valamint a gyorsítótárazást. Ez utóbbi arra szolgál, hogy a szekvencia ne szaladgáljon mindig a merevlemezhez, ha valaki új azonosítót kér tőle, hanem memóriából adjon neki egyet.

Egy INT típusú, egytől induló, egyesével növekedő szekvencia létrehozása így néz ki:

```
CREATE SEQUENCE IntSzamlalo AS INT
START WITH 1 INCREMENT BY 1
```

A következő lépés, hogy kérjünk egy azonosítót a szekvenciaobjektumtól. Ezt minden további nélkül megtehetjük az INSERT-től akár teljesen függetlenül is, így:

```
SELECT NEXT VALUE FOR IntSzamlalo
```

Az így kért értékek most elvesznek, mert nem csinálunk velük semmit. Egy INSERT-be így tehetnénk be:

```
INSERT Akarmi(szam)
VALUES(NEXT VALUE FOR IntSzamlalo)
```

Ezáltal a megkapott új értéket azon nyomban beletöltjük egy mezőbe, és el is mentjük.

8.3 DELETE helyben, kapcsolódó tábla alapján

A következő adatmódosítási utasításunk a DELETE. Azért ezt választottam, mert ez is, akárcsak az INSERT, komplett sorokkal dolgozik, merthogy fél sort törölni nem lehet.

A DELETE az az utasítás, amivel a rutintalan adatbázisgazda egyszer s mindenkorra megszabadul a vevőtörzstől, mert elfelejtett WHERE feltételt írni a törléshez. Merthogy az alap szintaxis mindent visz az adott táblából:

```
DELETE FROM Cities
```

Még az a szerencse, hogy készítettünk referenciális integritási szabályokat, és a városoknak van gyermekrekordjuk, ez a törlés nem csinál semmit. Ha csak egy gyermek is van, az egész tranzakció borul, törlés nem történik. Szerencsére.

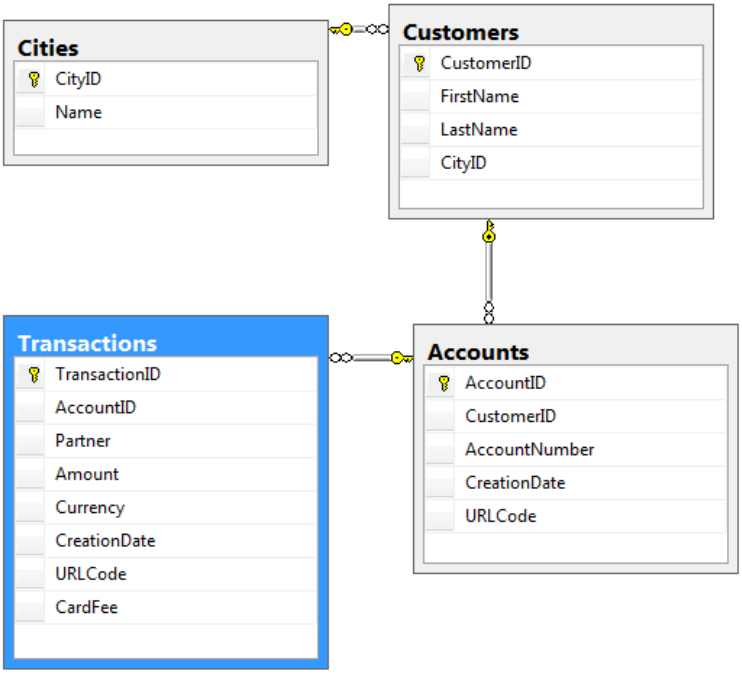
Egy adott sort törölni egy megfelelő WHERE-feltétel utánaírásával lehet, amit nem is részleteznék, mert olyat már láttunk.

A törlés igazi szépségét az adja meg, hogy törölni nemcsak egy adott sor értékeire készített WHERE-feltétel alapján tudunk, hanem a „szomszédos” táblákban megbúvó értékek alapján is. Értelemszerűen ez gyermekrekordok törlését jelenti az apa valamilyen tulajdonsága alapján, mert apát továbbra sem lehet törölni, amíg gyermekei vannak[17].

Az „apám alapján törlődöm” típusú törléshez úgy jutunk, hogy a DELETE-utasításba beleteszünk egy JOIN feltételt is. Amit az alap, szabványos DELETE nem tud. Akinek nem Microsoft SQL Servere van, az addig variál a WHERE feltételben mindenféle korrelált beágyazott lekérdezésekkel, amíg célt nem ér – vagy célt nem téveszt ugyebár. Nekünk azonban megadatott a kapcsolódás lehetősége oly módon, hogy a DELETE –utasításnak két FROM záradéka van. Az első kijelöli a törlés céltábláját, a második pedig létrehozza a kapcsolatot Apuval, és az alapján szűkíti a törlésre kerülő sorok számát.

Legyen az a feladatunk, hogy letöröljük az összes olyan tranzakciót, melyek olyan bankszámlához tartoznak, ahol a tulajdonos keresztneve Jakab! Vagy fordítva: töröljük Jakab összes tranzakcióját! Az előző írásmód a precízebb, mert abból kiderül, hogy nem a Customers, hanem a Transactions táblán kell indítani a törlést, majd két jól irányzott JOIN-nal fel kell kapaszkodni a Customerig, és megnézni, hogy Jakab-e. Emlékeztetőül az adatbázistérkép idevágó részlete:

[17] Kivéve azt az esetet, amikor valaki bekapcsolja a vízesésműveleteket az idegen kulcson, vagy újmagyarul: a kaszkádolást beállítja a foreign keyen.



45. ábra - tranzakció törlése két táblával odébb található érték alapján

A helyes utasítás így néz ki, figyeljük meg a dupla FROM-ot:

```
DELETE FROM Transactions
FROM Transactions t JOIN Accounts a ON t.AccountID=a.AccountID
JOIN Customers c ON a.CustomerID=c.CustomerID
WHERE FirstName='Jakab'
```

A DELETE-utasításnak is van OUTPUT kulcsszava, ha esetleg valaki el akarja kapni, hogy miket is törölt pontosan. A második FROM elé kell beírni, valahogy így:

```
DELETE FROM Transactions
OUTPUT DELETED.*
FROM Transactions t JOIN Accounts a ON t.AccountID=a.AccountID
JOIN Customers c ON a.CustomerID=c.CustomerID
WHERE FirstName='Jakab'
```

8.4 UPDATE helyben, kapcsolódó tábla alapján

A mesterhármas utolsó tagja az UPDATE, mely lehetővé teszi sorok, azon belül pedig egyedi mezők módosítását. Alapkiépítésben így néz ki, éppen átnevezem Gipsz Jakabot Kő Elemérre:

```
UPDATE Customers SET LastName='Kő', FirstName='Elemér' WHERE
LastName='Gipsz'
```

Mindjárt az első példám nagyon jól sikerült, mert megmutatja, hogyan kell egyszerre egynél több mező értékét is módosítani *(vesszővel felsorolva)*. Amit még mutatni érdemes, az egy olyan módosítás, amely a mező korábbi értékét használja fel az új érték kiszámítására. Emeljük meg 10%-kal az összes olyan tranzakció értékét, amelyeket 2012. január 20. előtt hajtottak végre!

```
UPDATE Transactions SET Amount=Amount*1.1 WHERE
CreationDate>'2012.01.20'
```

A DELETE megismerésével egyébként óriásit léptünk előre az UPDATE ismerete felé, mert a legbonyolultabb dolgot, a szomszéd tábla alapján történő módosítást már eleve tudjuk, hisz nyilván itt is további táblákat lehet a módosítóutasítás mögé csatlakoztatni egy FROM záradék után.

Ez így is van. Ha tehát szeretnénk extra bónuszt adni a piripócsi ügyfeleknek, akkor az alábbi négytáblás kapcsolatra lesz szükségünk, hisz a tranzakcióktól a város négy táblányira van:

```
UPDATE Transactions SET Amount=Amount*1.1
FROM Transactions t JOIN Accounts a ON t.AccountID=a.AccountID
JOIN Customers c ON a.CustomerID=c.CustomerID
JOIN Cities cc ON c.CityID=cc.CityID
WHERE Name='Piripócs'
```

És ha még a szokásos OUTPUT kulcsszóval is felszereljük, mindent fog tudni. Korábban említettem, hogy az UPDATE-nek két memóriatáblája van, az előtte (DELETED) és az utána (INSERTED) táblák.

```
UPDATE Transactions SET Amount=Amount*1.1
OUTPUT DELETED.*, INSERTED.*
FROM Transactions t JOIN Accounts a ON t.AccountID=a.AccountID
JOIN Customers c ON a.CustomerID=c.CustomerID
JOIN Cities cc ON c.CityID=cc.CityID
WHERE Name='Piripócs'
```

Ha valaki szeretne rémisztőt látni, nézze meg ennek az utasításnak a végrehajtási tervét! Erősen kilóg a képernyőről!

8.5 TRUNCATE

Az adatmódosító utasítások közé tartozik, így a teljesség igénye érdekében megemlítem a mi adatbázisunkban egyébként használhatatlan gyorstörlést, a TRUNCATE-t. Abban különbözik a DELETE-től, hogy nincs WHERE-feltétele, tehát mindig mindent visz. Ráadásul a műveletről naplóbejegyzés sem készül (lásd néhány bekezdéssel lejjebb), így gyors, mint a villám. Csak az az egy baja van, hogy nem lehet használni olyan táblákon, amelyikre idegen kulcs mutat. És melyikre nem mutat?

8.6 Tranzakciónaplózás

Biztosan van az olvasók között olyan, aki még emlékszik a DBase/Clipperes világra *(Clipper Summer '87!)*, és ha így van, talán az is beugrik, hogy a Clipperes alkalmazásokban előkelő helyet foglalt el a „szétesett adatbázis helyrepofozása" vagy valami hasonló elnevezésű menüpont. Azok az adatbáziskezelők rendszeresen maguk alá csináltak. Vajon az SQL Serverrel nem fordulhat elő ugyanez?

Elméletileg igen, gyakorlatilag nem.

Clipperéknél az volt a baj, hogy minden módosítás ész nélkül nekiesett az éles adatbázisnak, aztán ha a takarító néni kirúgta a kábelt, akkor a módosítás szépen félbemaradt. Az SQL Server úgy védekezik az áramkimaradásból és egyéb katasztrófákból származó adatvesztések ellen, hogy soha, de soha nem írja egyenesen az éles adatterületet, hanem ehelyett minden módosítási műveletet az úgynevezett tranzakciónapló fájlban rögzít, és ha ez sikerrel zárul, akkor fog nekiesni az éles adatoknak. De ekkor már nincs veszély, mert még ha az áttermelés félbe is marad, van egy 100%-ban leírt változat a naplóban, ami alapján a megszakadt írási műveletet folytatni lehet.

Ha a naplóírás félbeszakad, annyi baj legyen, az éles adatterület nem módosult, a log meg csak log, van benne egy kis szemét és kész. Ha azonban a naplóírás sikeres, előbb-utóbb az éles adatterületre is átvezetésre kerül a módosítás, akárhogy kapálódzik ellene a partvissal a takarítónő.

Úgy is mondhatjuk, hogy az SQL Server mindent kétszer ír, kétszer ír.

Ami persze teljesítményveszteséggel járna, ha a második írást is megvárnánk, mielőtt az írás eredményét a felhasználók láthatnák. De igazából már a naplóírás sikeres lezárulása után elérhető az új adat, mert az SQL Server trükközik az adatokkal a memóriában.

A módosítások első fázisában, amikor a naplóírás történik, akkor van bent a boszorkány a konyhájában, és a naplóba irkafirkál. Ebbe ekkor még senki nem láthat bele. Amint azonban a módosítás a naplóban sikerrel zárult, a memóriában azonnal az új változat lesz olvasható mindenki számára, míg az éles adatterület felülírása szép csendben megtörténik a háttérben.

Hol érdekel ez minket?

8.7 Implicit és explicit tranzakciók

Hát ott, hogy már látjuk, vannak bizonyos munkaegységek, amelyek vagy egy az egyben megtörténnek, vagy egyáltalán nem. Félbehagyás esetén nem történik változás az eredeti adatokon.

Ezt a munkaegységet hívjuk tranzakciónak, és az SQL Server adatbázisok írásakor az az alapértelmezett működés, hogy a tranzakciókat elsőként a tranzakciónapló fájlba írjuk, és ha ott végleges… de most önmagamat ismétlem. Ezzel vertük agyon a DBase/Clipper hegemóniát *(no meg az adatbázismotorral…)*.

Szóval van ez a bizonyos munkaegység, a tranzakció. Amiről azért érdemes tudni, mert bár automatikusan működik az egyes adatmódosító utasításokra *(implicit tranzakció)*, valójában mi magunk is használ vehetjük ennek a működésnek azáltal, hogy össze tudunk fogni egy sereg egymással összefüggő módosítást, és azt tudjuk kérni az SQL Servertől, hogy ezeket lécci-lécci vagy egy az egyben csináld meg, vagy ha félbeszakad a művelet, akkor töröld, mintha semmit sem csináltunk volna.

Tehát ha leírunk egy akármilyen adatmódosító utasítást, akkor ott lefut, és sikeresen záródik is egy tranzakció? Igen, így van. Minden egyes adatmódosító utasítás a háttérben valójában így fut le:

```
BEGIN TRAN
INSERT Cities(Name) VALUES('Alma Ata')
COMMIT TRAN
```

Csak éppen a BEGIN TRAN, COMMIT TRAN utasításokat nem kell kiírni, azok alapértelmezés szerint vannak. Implicite.

Ha finoman szeretnénk szabályozni a tranzakciók elejét és végét, akkor ki kell írnunk, és ez a módja annak, hogy sok-sok utasítást egy csomagba fogjunk össze. Vegyük az alábbi kereskedelmi szituációt:

```
INSERT Szamlatetel(...) VALUES(...)
UPDATE Raktarkeszlet SET Mennyiseg=Mennyiseg-1
      WHERE...
```

Itt nem jó, ha csak az egyes utasítások hajtódnak végre 100%-os bizonyossággal, mert ha a két utasítás közül az első igen, a második pedig nem, akkor a raktárkészlet több elemet fog tartalmazni, mint amennyi a valóságban a raktárban van, és kész a leltárhiány.

Ha a két sort felcseréljük, akkor sem lesz jobb a helyzet, félbemaradás esetén csökkentjük a raktárkészletet, de mégsem adjuk el, tehát fantomtétel keletkezik a raktárban, amit a dolgozók büntetlenül hazavihetnek, mert a „gép” nem tud róla.

Az ilyen esetek kivédésére való a tranzakciók elejének és végének kézi megadása. Mert ebben az esetben mi határozzuk meg, hogy a boszorkány mikor jöhet ki a kamrájából, mikor végzett a mágiával. A fenti eladási szitu így néz ki helyesen:

```
BEGIN TRAN
      INSERT Szamlatetel(...) VALUES(...)
      UPDATE Raktarkeszlet SET Mennyiseg=Mennyiseg-1
      WHERE...
COMMIT TRAN
```

És ez már valódi „amíg a vonat az állomáson tartózkodik” életérzés!

8.8 Mitől "savas" egy tranzakció?

A nagyon okos adatbázisguruk azt szokták mondani a tranzakciókról, hogy savasak (ACID). Ez a következő dolgokat jelenti a gyakorlatban:

- Atomic. Egy tranzakció attól atomi, hogy egy és oszthatatlan. A benne foglalt utasítások együtt érnek célba, vagy együtt hullanak a sírba.
- Consistency. A konzisztencia itt azt jelenti, hogy a tranzakció az adatbázist egy korábbi, érvényes állapotból egy új, de szintén érvényes, önellentmondásmentes állapotba viszi. Vagyis nem hagy félbehagyott alkatrészeket maga után.
- Isolation. Az elzártság a boszorkánykonyhára utal. Amíg kotyvasztunk, addig abba másnak semmi beleszólni, beleszagolnivalója nincs. Őket csak a késztermék érdekelje!
- Durable. Tartósság. Ami megtörtént, megtörtént. Ami COMMIT paranccsal zárult, az kivétel nélkül végleges módosítás.

8.9 BEGIN TRAN, COMMIT, ROLLBACK

Fentebb már láttuk, hogyan lehet egy tranzakciót kézzel indítani (BEGIN TRAN), illetve sikeresen lezárni (COMMIT). Van azonban a sikertelen lezárásnak is hivatalos útja, ez pedig a ROLLBACK TRAN. Ha egy utasítássorozat kellős közepén kiderül, hogy a tranzakció véglegesítésének feltételei nem adottak, mert mondjuk üres a raktár, vagy épp piros hó esik, a tranzakció kinyírhatja önmagát egy ügyesen elhelyezett ROLLBACK utasítással.

Ha valaki alaposan ismeri a tranzakciók belső feldolgozását, akkor ilyet nem csinál, mert a ROLLBACK iszonyú költséges művelet[18].

Az explicit tranzakciók kapcsán még egy fontos tényező a tranzakciók egymásba ágyazása. Ha egy BEGIN TRAN-ban elindítok egy BEGIN TRAN-t, akkor már két tranzakció dübörög egymásban, és ezeket látszólag külön vissza tudom vonni, ha akarom. A valóság azonban más. Az alábbi kódrészlet rávilágít a valódi működésre, ahol a @@TRANCOUNT globális változó mindig megmutatja, éppen hány tranzakció kupacolódott egymásra:

```
BEGIN TRAN
SELECT @@TRANCOUNT --Egy
BEGIN TRAN
SELECT @@TRANCOUNT --Kettő
ROLLBACK
SELECT @@TRANCOUNT --Nulla
```

Mint az látható, egy darab ROLLBACK kinyírta mindkét tranzakciót. Egy nyisszantással elvágta a torkukat. Ez így működik, erre így kell számítani[19].

Ne menjenek el, a tranzakciók működésének élveboncolását a második tuningfejezettel folytatjuk.

[18] Merthogy technikailag ilyenkor nem az én tranzakcióm gördül vissza, hanem az általam érintett 8k-s lapon mindegyik rekord időutazást szenved vissza a múltba, majd rajtam kívül mindenki más előregurul, én meg úgy maradok. Ebből már érthető, hogy minél többen dolgoznak párhuzamosan, a ROLLBACK annál szörnyűbb teljesítményt nyújt.

[19] Geekeknek: igen, lehet ezzel még tovább kínlódni, SAVE TRAN meg ilyenek, de azt amúgy a kutya sem használja. Mert senki nem érti. ☺

9 Tuning alapok II.

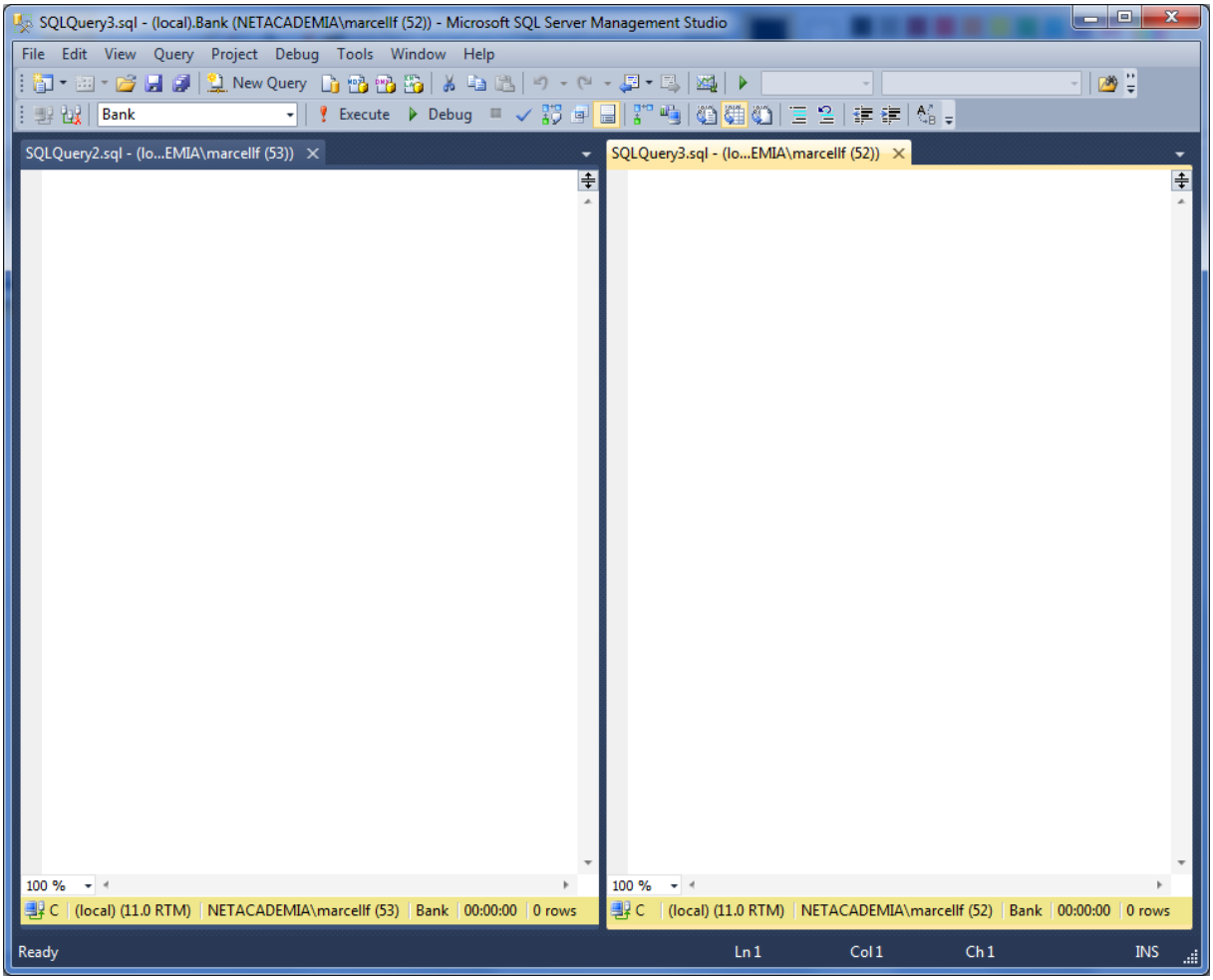
Az egyik legszebb teljesítménytuningolós feladat, amivel szaki szembekerülhet, az álló SQL Server, melyen minden folyamat beragadt. A telefonok csörögnek, „nem lehet számlázni”, ugyanakkor a processzorterheltség 0%, a memória üres, a hálózat csendes, a merevlemez nem forog. Na, ilyenkor mi a teendő?

Sok olyan céggel találkozunk, ahol ebben a nyilvánvaló helyzetben – vesznek egy nagyobb vasat. Az igaz, hogy az előző sem volt leterhelve, de hát állt az alkalmazás, nem? Akkor erő kell alá! Az egyik Kedves Vevőnk odáig fajult, hogy három hullámban vett nagyobb gépeket, mire elgondolkodott a jelenségen, és hozzáértő tanácsát kérte.

Ha laborkörülmények között elő tudnánk állítani ilyen helyzetet, könnyebben megértenénk, mi is zajlik ilyen esetben az SQL Server lelkében. Jó hírem van, elő tudunk állítani, mégpedig igen könnyedén!

9.1 Zárolási rendszer, livelock, deadlock, sp_lock

A most következő kísérletekhez két egyidejű kapcsolatra lesz szükségünk. Nyissunk meg két lekérdezőablakot, majd ízlésesen rendezzük őket egymás mellé a Window/New Vertial Tab Group menüpont segítségével! Valami ilyesmit kell látnunk:



46. ábra - kétablakos elrendezés

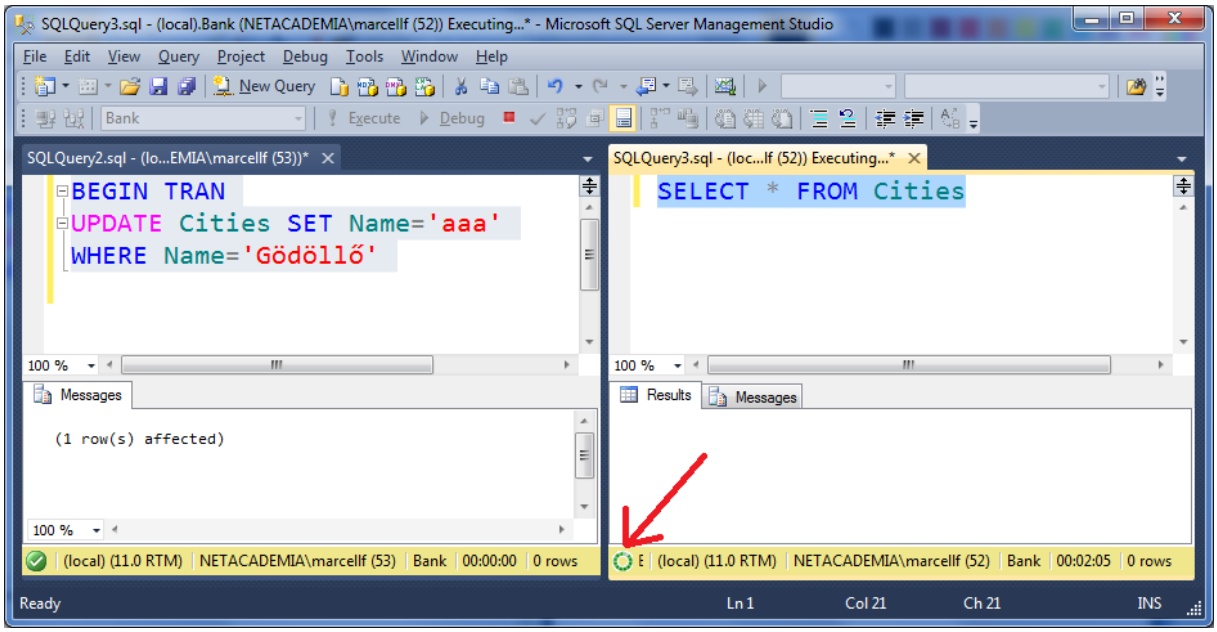
Ez a két ablak fogja reprezentálni a sokezer párhuzamos kapcsolatot. A bal oldaliba írjuk be az alábbi félkész tranzakciót, majd futtassuk le:

```
BEGIN TRAN
UPDATE Cities SET Name='aaa'
WHERE Name='Gödöllő'
```

A tranzakciót szándékosan nem zártam le, hiányzik a végéről egy COMMIT vagy egy ROLLBACK. A másik ablakba pedig kerüljön ez a lekérdezés:

```
SELECT * FROM Cities
```

Ez is fusson! Kiemelem, hogy mit kell látni.



A lekérdezés teljesítménye némi kívánnivalót hagy maga után, noha csak egy ötsoros táblán dolgozik. Jöhet a méricskélés, vajon mi kevés, és kiderül, hogy semmi sem kevés. Itt egy tipikus zárolási problémával állunk szemben, a banya nem kompatibilis a hallal. Szakmai nyelven szólva a bal oldali tranzakció által a táblán (?) elhelyezett kizárólagos zár (exclusive lock) megakadályozza, hogy az olvasást végző műveletek félkész adatokat olvassanak. I, mint izoláció. (ACID)

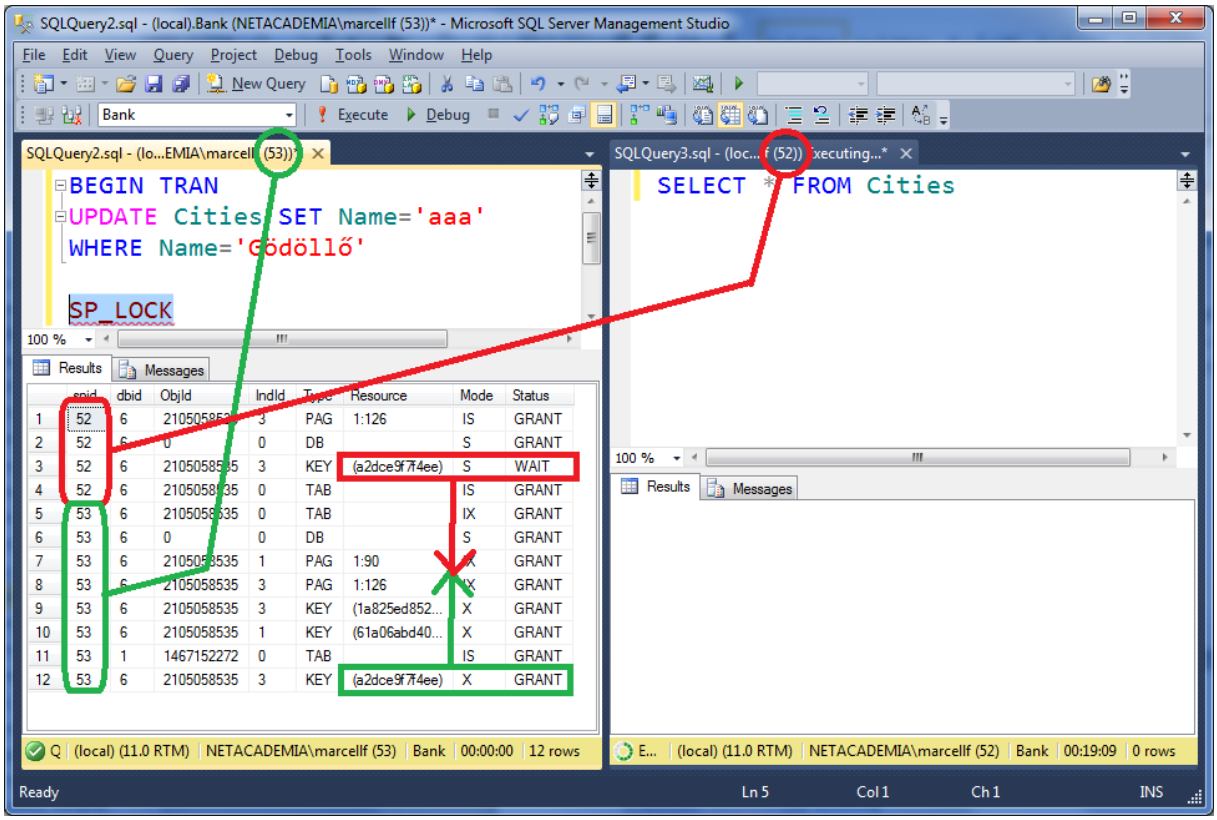
Meddig áll fenn ez az állapot? Ameddig a bal oldali tranzakció véget nem ér. Lehet, hogy ez órákig tart? Lehet. Lehet, hogy percekig? Az is lehet. És hányszorosára gyorsítja az a guru az alkalmazást, aki megszünteti a zárolási ütközést? Akár végtelenszeresére.

Most persze mondhatnánk, hogy ilyen a valóságban nincs. Egyrészt igenis van, léteznek ennyire idióta alkalmazások, másrészt ez a kísérlet csupán modellezi, hogy mi történik, ha sok tízezer, egyenként milliszekundumos tranzakció fekszik egy lekérdezés útjába. Ugyanez történik.

Miért nem lát egyetlen sort sem a lekérdezés, noha a tranzakció csupán egy sort módosít? Nos, azért nem, mert az indexelésünk jó ugyan, de iciri-piciri a tábla, ezért pontosan ugyanazokért a sorokért

versengünk. Éles környezetben, sok adattal és sok felhasználóval azt látnánk, hogy a lekérdezés döcögve fut le, de lefut. Meg-megakad az éppen módosított sorok előtt, és teljesen kiszámíthatatlan, mennyi idő alatt ér véget, de végül csak lefut.

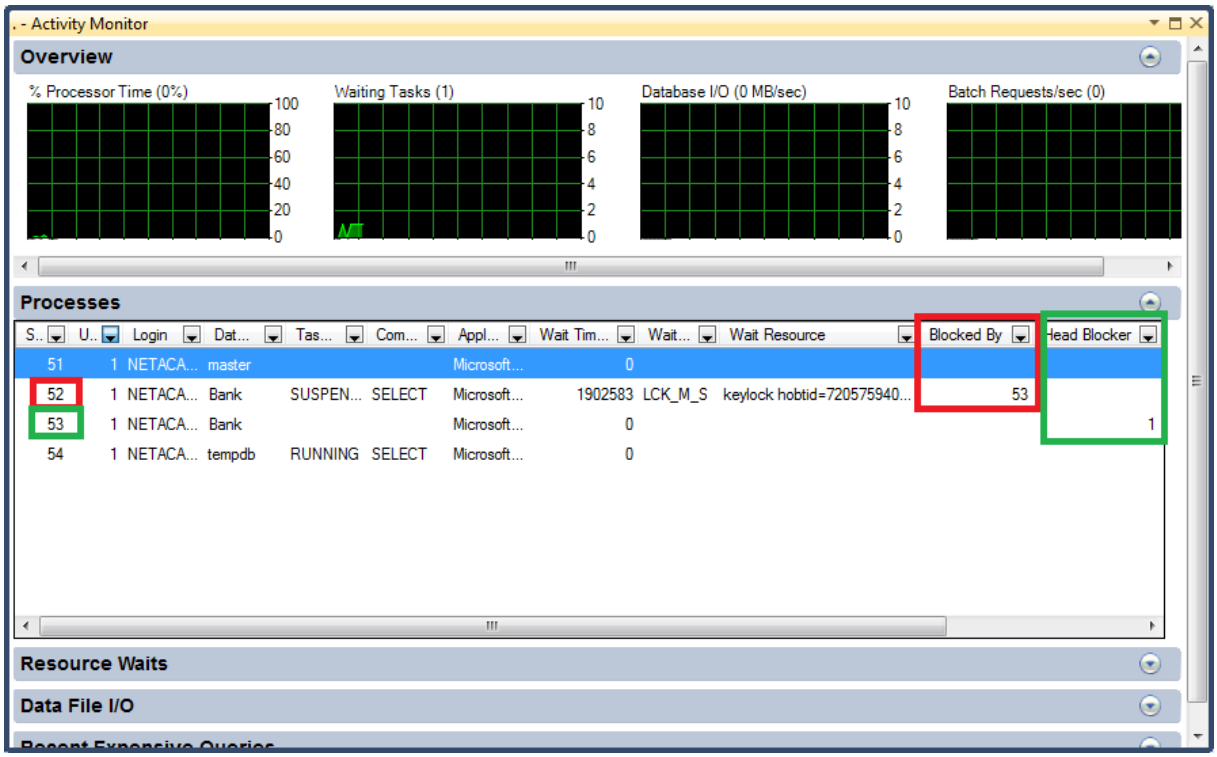
A jelenség alaposabb megismeréséhez a zárolási rendszer ismerete visz közelebb. Van egy jó öreg tárolt eljárás, amit én a zárak kilistázására használok 1873 óta, úgy hívják, sp_lock. Tudom, van ennél modernebb dm ugyanennek a lekérdezésére, de az sp_lock olyan kis kompakt táblázatot tár elénk, hogy nem tudok leszokni róla. A bal oldai ablakunk él, ott futtassuk le!



47. ábra - Milyen zárak vannak érvényben?

Ötvenketteske és ötvenhármaska dolgozik vadul. A táblázatból kiolvasható, hogy mindketten a hatos számú adatbázisban vannak főerővel *(ez az én Bank adatbázisom)*, emellett ötvenhármaska a Masterben is kavar valamit *(egyes adatbázis)*, de ezt most nem elemezzük. Az Objid oszlop is eléggé egysíkú, ami arra vall, hogy ugyanazt az objektumot macerálják, ami valójában a Cities tábla. A legutolsó oszlopban pedig látszik a sok GRANT között egy WAIT. Ötvenketteske várakozik egy S *(Shared, technikailag Select)* típusú zárra a SELECT megkezdéséhez az a2dce… kezdetű soron, de ugyanezen áll ötvenhármaska is, neki pedig egy X *(exclusive)* zárja van ugyanerre.

A jelenséget megnézhetjük az Activity Monitorban is.



48. ábra - zárolási probléma az Activity Monitorban

Mit lehet ilyenkor tenni? Az Activity Monitorral például ki lehet lőni a Head Blockert (=főbűnös), de akkor az ő tranzakciója megsemmisül. Valamint ki lehet várni, hogy ez az igencsak hosszúra nyúlt tranzakció befejeződjön. Adjunk ki a bal panelen egy ROLLBACK utasítást, mire a jobb panel ott helyben megtáltosodik.

A naiv megközelítések *(még több RAM, még több processzor)* ebben az esetben nem vezetnek eredményre. Ez az alkalmazás kukára való. De azzal együtt is, hogy szemétre való, lehet, hogy csak ez jutott nekünk, nincs cserelehetőség, tehát tovább kell nyomulnunk az esetleges megoldás felé.

9.2 Szemétolvasás, optimizer hintek

Összeakadgató alkalmazások ellen többféle gyógyír is alkalmazható. A hagyományos megközelítéssel kezdjük, és onnan haladunk az ultraszuper megoldások felé.

Az első megközelítés lényege, hogy benézünk a boszorkánykonyhába. Egyszerűen beleolvasunk abba, ami még kész sincsen. És mivel nincs készen, és lehet, hogy soha nem is lesz, ezt a technikát szemétolvasásnak *(dirty read)* hívják. Az eljárás lényege, hogy a SELECT-nél felveszünk egy olyan optimizer hintet, ami a zárak figyelmen kívül hagyására ösztönzi a lekérdezést.

```
SELECT * FROM Cities WITH (NOLOCK)
```

És már olvassuk is akadálytalanul a szemetet, az ’aaa’ nevű várost, ami soha nem lesz, mert a tranzakció végül ROLLBACK-ra fog futni. Viszont a „lefagyás” megoldódott, mégpedig anélkül, hogy nagyobb vasat vásároltunk volna.

Ez szép és jó *(illetve rossz és csúnya, nézőpont kérdése)*, de ha ezt minden döcögős lekérdezésbe bele kell írni, abba belehalunk. Nem beszélve azokról a lekérdezésekről, amelyek bele vannak égetve a BANK.EXE-be, azokat hogyan írjuk át? Sehogy. De nem baj, van máááásiiik!

Ugyanezt a hatást elérhetjük a kapcsolat izolációs szintjének megváltoztatásával, anélkül, hogy a lekérdezést módosítanunk kellene.

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
```

Ez az utasítás a normaszegést teszi alapértelmezetté, innentől minden lekérdezés szemétolvasóvá válik ezen a kapcsolaton. Át tudjuk-e állítani szemétolvasóssá a BANK.EXE kapcsolatait? Talán igen, talán nem. De nem baj, van máááásiiik!

A harmadik megközelítés lényege, hogy az egész zárolási rendszert egy másik üzemmódra állítja be. Boszorkánykonyhát, elszigeteltséget nyitni nemcsak zárolással, hanem verziózással is lehet. Ebben a modellben az történik, hogy ha valaki módosít egy adatot, akkor annak egy új példányát, egy másolatát bütyköli, a többiek pedig továbbra is olvashatják az előző verziót. Amikor a bütykölés készen van, az új verzió lép életbe, a régi pedig leköszön.

A módszer hátránya, hogy csak olyan helyen tudjuk bekapcsolni, ahol hozzáférünk az adatbázishoz, tehát felhőben nem. Előnye viszont, hogy az alkalmazás maradhat olyan buta, amilyen, mert többé nincs zárolás. Óriási előny, hogy szemétolvasás sincs, mert nem szemetet, hanem az adat előző változatát tudjuk olvasni.

A mi helyi adatbázisunk a mi kezünkben van, nosza, kapcsoljuk be a verziózást! Ha mindenkit kizavarunk az adatbázisból (tippek, trükkök erre x oldallal feljebb olvashatók), az alábbi parancs váltja át a Bank adatbázist sorverziósra:

```
ALTER DATABASE Bank
SET READ_COMMITTED_SNAPSHOT ON
```

És mostantól mindenki boldog[20].

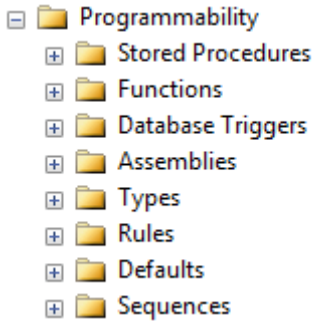
Valójában a zárolások témaköre önmagában egy egész kötetet tenne ki, amitől most nagyvonalúan eltekintünk. És persze tranzakcióizolációs szintből sem egy van, hanem öt. De mennünk kell tovább.

[20] Csak a tempdb nem…

10 Programozás

Mind ez ideig halmazokkal dolgoztunk, akár lekérdezésekről, akár adatmódosításról volt szó. A TSQL-nyelv azonban ismeri a programozás fogalmát is, és bizony nem nehéz olyan feladatba belefutni, ahol szükség lenne a lépésenkénti futtatásra, ciklusra, elágazásra, változókra. Elég, ha valamilyen bonyolultabb üzleti logikára gondolunk, amelyet nem lehet egy SELECT utasításban megvalósítani, és máris a programozásnál vagyunk.

Emellett ide tartozik az SQL terminológiája szerint minden tárolt eljárás, függvény és sok minden más, amelyek az adatbázisunkban a Programmability ágban csücsülnek a fán.



49. ábra - programozható akármik az SQL Serverben

10.1 Változók

A FLOAT-os végtelen ciklusnál használtunk már változót, így most csak hivatalossá teszem a megtörtént eseményeket. A változókat DECLARE kulcsszóval vezetjük be, nevük előtt @ (kukac) karakter áll, adattípusukat tekintve pedig az összes korábban felsorolt adattípust felvehetik, sőt többet, mert lehet készíteni tábla típusú változót is. Tipikus felhasználási területük a ciklusváltozó, vagy valamilyen más átmeneti adat tárolása, és szerepelhetnek függvényekben, tárolt eljárásokban és sima scrpitekben.

Mivel a hagyományos változók létrehozásának poénjét sok-sok oldallal korábban lelőttem, itt és most egy táblatípusú változót hozok létre, adatok átmeneti tárolására.

```
DECLARE @t TABLE(
      ID INT PRIMARY KEY IDENTITY NOT NULL,
      Name NVARCHAR(88),
      BirthDate DATETIMEOFFSET NOT NULL DEFAULT '1999.11.11'
)
```

Mint látható, nem ördöngösség, sima következtetés segítségével ki lehet találni a szintaxist, mivel megegyezik a CREATE TABLE szintaxisával. A deklarálást követően pedig úgy lehet használni, mint egy táblát, melynek neve @t.

```
INSERT @t(Name, BirthDate) VALUES('Micimackó', '1926.04.17')
```

Csacsi, öreg medvém!

Van egy másik hasonló konstrukció, az átmeneti (temp) tábla, ami ugyanez, de mégis más. Ő nem kukaccal kezdődik, hanem szőnyeggel (#), és nem deklaráljuk, hanem CREATE TABLE kell hozzá, és nem füstöl el magától a script végén, hanem DROP TABLE utasítással kell ledobni. Ezek a szemmel látható különbségek. Íme ugyanez a feladat temptáblával:

```
CREATE TABLE #t(
ID INT PRIMARY KEY IDENTITY NOT NULL,
Name NVARCHAR(88),
BirthDate DATETIMEOFFSET NOT NULL DEFAULT '1999.11.11'
)
```

A választást megkönnyíti, ha tudjuk, hogy - ha lehet ilyet mondani - a temptábla „táblább” a táblatípusú változónál. A temptábla például tranzakcionált, indexelhető, és nagy adatmennyiségeknél sebességtesztekben veri a táblatípusú változót. A táblaváltozó meg kicsi és ügyes.

Meg kell említenünk még a globális változókat, melyek neve két kukaccal kezdődik (@@), van a rendszerben vagy negyven ilyen (például @@TRANCOUNT), de mi magunk nem tudunk globális változót létrehozni. Aki azt állítja, létre tud hozni globális változót, mert beírja a neve elé a két kukacot, és kész, azoknak tetszeni fog ez a példa, mert én akkor ezek szerint csináltam nagyonglobális és hiperglobális változókat.

```
DECLARE @@b INT
DECLARE @@@b INT
DECLARE @@@@b INT
```

De nem. A többletkukacok a változónév részét képezik csupán.

És vannak globális temptáblák, duplarétegű szőnyeggel (##), amelyeket mi hozhatunk létre, és az egyszőnyeges társaikkal ellentétben más felhasználók is elérik, nem csak mi.

A változók legfontosabb felhasználási területe mégiscsak a…

10.2 Ciklus, elágazás

Ciklusból szerényen egy fajta van, a WHILE. Nemcsak, hogy több a semminél, de tökéletesen elegendő minden feladatra, mert ha valakinek FOR ciklusra van szüksége, le tudja vele programozni. Csak egy ciklusváltozót kell kézzel állítgatnia és kész.

Egymillió sort akár így is generálhattunk volna:

```
DECLARE @i INT
SET @i=1000000
WHILE @i>0
BEGIN
      SET @i=@i-1
      INSERT ...
END
```

Elágazásból pedig itt van nekünk a csacsi, öreg IF, aki minden informatikusnak a könyökén jön ki. Egy gyors példa a teljesség kedvéért, ennél értelmesebb példát majd a tárolt eljárásoknál használunk:

```
DECLARE @i INT
SET @i=1000000
IF @i=1000000
BEGIN
	PRINT 'Egymillió'
END
ELSE
BEGIN
	PRINT 'Nem egymillió'
END
```

10.3 Eset-leg (CASE)

A CASE utasítás már érdekesebb. Merthogy ez nem csupán az, mint amire emlékszel a Commodore 64 BASIC alapján. Ez nemcsak egy elágazás (bár az is tud lenni), hanem kifejezések helyén szerepelhet SQL-utasításokban, így alkalmas például arra, hogy 0/1 értékeket röptében kicseréljen Férfi/Nőre. Micimackóval eljátszva ugyanezt

```
DECLARE @t TABLE(
	ID INT PRIMARY KEY IDENTITY NOT NULL,
	Name NVARCHAR(88),
	BirthDate DATETIMEOFFSET NOT NULL DEFAULT '1999.11.11',
	Female BIT
)

INSERT @t(Name, BirthDate) VALUES('Micimackó', '1926.04.17')

SELECT Name, CASE Female WHEN 0 THEN 'Férfi' WHEN 1 THEN 'Nő' ELSE
'Plüss' END AS Sex
FROM @t
```

Nos, hősünk a plüssállat nembe tartozik, mivel a nemi kérdést egy nullozható bitben oldottam meg, és nem töltöttem ki az INSERT-nél, tehát se nem 0, se nem 1.

10.4 Tárolt eljárás

A kedvenc programozási eszközöm a tárolt eljárás. nemcsak azért, mert „mindent tud”, hanem mert orrán-száján dől az adat, ezért igen sokoldalúan lehet felhasználni. Emellett van egy olyan kedvező tulajdonsága *(ami azért néha visszaüt)*, hogy nem fordítódik le minden futtatáskor, hanem a végrehajtási terve és kódja is gyorstárazásra kerül, tehát ha ezt hívom másodpercenként tízezerszer, nyerek másodpercenként tízezer szintaktikai ellenőrzést, objetumhivatkozás-feloldást, végrehajtási terv készítést és fordítást, amin egyébként egy sima SQL-lekérdezés mindig átmegy.

Szóval gyors. Nagyon gyors. Lehet. *(Ha az indexeléssel és csillagos ötös lekérdezésekkel alá nem vágunk.)*

A tárolt eljárás tetszőleges számú Transact SQL utasítást és tetszőlegesen bonyolult kódot tartalmazhat. Meghívásakor bemeneti paramétereket adhatunk át neki, amit ő ügyesen feldolgoz. Adatot visszajuttatni pedig a következő módokon tud *(orrán-száján)*:

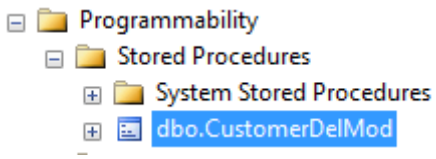
- Szerepelhet benne SELECT, aminek az eredménye a hívó oldalán szépen felbukkan. Egynél több SELECT esetén egynél több eredményhalmazt kapunk, csak győzzük feldolgozni!
- Van neki visszatérési értéke, ami ugyan csak egy INT, de akkor is, egész számokkal jól lehet üzengetni a hívónak.
- Tud cím szerinti paraméterátadást, ami azt jelenti, hogy a bemeneti paramétereit átírva a hívónál is megváltoznak az eredeti értékek.

Paraméterezhetőségével megszégyeníti a nézeteket, amelyeknél erre nincs lehetőség, azok csupán „táblák", WHERE-feltétellel. A paraméterezhetőség egyben csapda is, a Programozó Legnagyobb Tévedései gyűjteménybe való, mert ha valaki egy bemeneti paraméterrel… á, inkább lássunk egy példát! Egy ROSSZ, de elterjedt példát.

Az alábbi BETEG tárolt eljárás a bemenő paraméterének függvényében vagy töröl a Customers táblából, vagy módosít a rekordon. A lusta programozó egyetlen tárolt eljárással szeretne minden funkciót megoldani egy táblán.

```
CREATE PROCEDURE CustomerDelMod @CustomerID INT, @Method INT,
@LastName NVARCHAR(88)
AS
IF @Method=1
BEGIN
     DELETE Customers WHERE CustomerID=@CustomerID
END
ELSE
BEGIN
     UPDATE Customers SET LastName=@Lastname WHERE
CustomerID=@CustomerID
END
```

A kész tárolt eljárás becsücsül a fába:



50. ábra - tárolt eljárás az Object Explorerben

A @Method bemeneti paraméter változtatásával lehet a futást az egyik vagy a másik ágra terelni. Ez a tárolt eljárás akkora katasztrófa, hogy meg sem mutatom, hogyan kell meghívni, hátha akkor nem írja meg így senki. A hiba oka nem az, hogy a benne lévő logika ne lenne jó, mert az hibátlan. Ámde

fentebb megállapítottuk, hogy a tárolt eljárás az első futásakor értékelődik ki, és készül hozzá végrehajtási terv és kód, ami bent marad a memóriában, későbbi futtatás céljára.

Igen ám, de ha az első híváskor mondjuk törlünk, akkor a törlésre vonatkozó végrehajtási terv készül és memorizálódik, amellyel talán lehet sorokat módosítani is, de hogy milyen teljesítménnyel, azt jobb nem firtatni.

Na jó, csak elárulom, hogyan lehet megmenteni az ilyen hatlövetű tárolt eljárásokat. Egyszerű! Csak el kell vetni a tárolt eljárás legfőbb előnyét, az elmentett végrehajtási tervet! Ha minden hívásnál mindig lefordítjuk, nincs kesselési probléma. Van viszont sok-sok fordításunk, tehát végül ezzel a megoldással nem nyerünk semmit. A sima hívás így fest:

```
EXEC CustomerDelMod 11, 1, 'Imre'
```

Az állandóan újrafordítgatós pedig így:

```
EXEC CustomerDelMod 11, 1, 'Imre' WITH RECOMPILE
```

Ezzel el is árultam, hogy a tárolt eljárások végrehajtásának utasítása az EXEC, amit néha nem látunk leírva. El lehet hagyni az EXEC-et abban az esetben, ha a tárolt eljárás hívásunk egy batch legeslegelső utasítása. Minden más esetben, valamint a korrektségre és áttekinthetőségre való törekvés jegyében ki kell írni.

Ha egy *(vagy több)* SELECT szerepel a tárolt eljárás törzsében, akkor egy *(vagy több)* eredményhalmazt is produkál. Ez azért nagyon jó, mert így az ügyfélprogramokban lehetőség nyílik a táblák helyett egy–egy tárolt eljárást megadni adatforrásnak, ráadásul bemeneti paraméterekkel. Még ma is divatos az a programozási megközelítés, hogy egy ügyfélprogramban minden adatművelet helyett tárolt eljárást adnak meg, mert kisebb, gyorsabb és még biztonságosabb is.

Az adathalmazon felül visszatérési értéket is produkálhat a tárolt eljárás, ezzel tud üzenni a hívónak, hogy például sikeres vagy sikertelen volt-e a futása, vagy hogy nyár van-e vagy tél. Merthogy tetszőleges értelmet adhatunk a visszatérési értéknek, ami viszont csak egy sima INT típusú szám lehet. Tehát ha nyár vagy tél, akkor mondjuk az 55 a nyár, a -77 a tél választ jelenti. Önkényesen.

```
CREATE PROCEDURE Evszak @Be DATETIME
AS
DECLARE @Honap INT
SET @Honap=DATEPART(MONTH, @Be)
IF  @Honap>5 AND @Honap<9
BEGIN
     PRINT 'Nyár'
     RETURN 55
END
IF  @Honap<3 OR @Honap>10
BEGIN
     PRINT 'Tél'
     RETURN -77
END
```

TSQL-scriptből a visszatérési érték kinyerése nem triviális, el kell nyeletnünk egy INT típusú változóval, amivel már azt csinálunk, amit akarunk. Ugyanez egy kliensprogramban csupán a @RETURN vagy @RETURN_VALUE nevű „bemeneti" *(merthogy a paraméterek között találjuk)* paraméter kiolvasását jelenti. TSQL-megoldás a problémára:

```
DECLARE @Visszateresi INT
EXEC @Visszateresi=Evszak '1999.06.06'
PRINT @Visszateresi
```

A harmadik adatvisszaadás a hívási paramétereken keresztül zajlik. Cím szerinti paraméterátadást ritkán használunk, mert valójában fáj. Rossz nézni, ahogy a hívó egyik változóját a hívott át tudja írni. De hát van ilyen, lássuk, hogy működik!

Cím szerintivé akkor válik a paraméterátadás, ha ezt mindkét fél egyértelműen akarja. A hívó is, és a hívott is. Erőszakoskodás nincs. Ha mindkét fél odateszi egy paraméter elé, hogy OUTPUT, akkor az értékadás cím szerintivé alakul, vagyis a változó memóriacímét adjuk át annak konkrét értéke helyett. És ha a hívott megkapja a változó címét, akkor arra a memóriacímre oda tud írni, így valósul meg a ketten szeretünk mi egyet. Ha bármelyik fél visszakozik, vagyis elhagyja az OUTPUT kulcsszót, visszaáll a rend az érték szerinti paraméterátadásra. Lássuk.

```
CREATE PROCEDURE Duplazo @Valtozo INT OUTPUT
AS
SET @Valtozo=@Valtozo*2
```

A tárolt eljárásunk készen áll a párosodásra, ha valaki azt mondja neki, hogy OUTPUT!, akkor ő is azt fogja válaszolni, hogy OUTPUT!

Hívjuk meg először simán:

```
DECLARE @Bemeno INT=3
EXEC Duplazo @Bemeno
PRINT @Bemeno
```

A bemenő értékünk három, melynek a duplája – ööö – három. Ez azért van, mert a hívó nem OUTPUT-os, és már ugrott is a cím szerinti paraméterátadás.

Ha ellenben így hívom meg:

```
DECLARE @Bemeno INT=3
EXEC Duplazo @Bemeno OUTPUT
PRINT @Bemeno
```

Akkor a három duplája hat lesz.

Most úgy tűnhet, hogy feltaláltuk a függvényt, pedig egy csudát. A bemenő paraméterek felülírkálása nem fogja a tárolt eljárásunkat kihúzni élete legnagyobb csávájából, hogy tudniillik nem lehet őt felhasználni kifejezésekben. Csak így lehet meghívni, EXEC-kel. SELECT utasítás mezőlistájában ezzel duplázni? Felejtsd el!

Most pedig készítsünk valami komoly tárolt eljárást a bankunkhoz! Egy olyan funkciót szemeltem ki, amit viszonylag ritkán, de következetesen és automatikusan kell lefuttatni, ez pedig a havi kamatszámítás. Nagyon fontos ennél az eljárásnál, hogy valahogyan naplózni kell, kinek csináltuk már meg a kamatszámítást és –jóváírást, hogy a folyamat a sokmillió ügyfelünknél tetszőleges ponton megszakítható, és adatvesztés, valamint kamatnyereség nélkül folytatható legyen.

Ez a naplózás nem a tranzakciónaplót jelenti, hanem egy alkalmazásszintű, általunk kezelt táblát, amiben vezetjük, hogy kinek, mikor, melyik számláján számoltuk ki a kamatot. Magát az összeget pedig visszavezetjük az ügyfél számlájára.

Ha naplózás, akkor kézi tranzakcióra is szükségünk lesz, mert micsoda naplózás az, amikor megcsinálom a kamatszámítást, jóváírom az ügyfélnél a nyereséget, de a naplómba nem írom be? Dupla haszon az ügyfélnél, ami megengedhetetlen.

Azt is előrebocsátom, hogy láttam én már hasonló tárolt eljárást élőben, és nem három paramétere volt, hanem ötven. Aki esetleg úgy gondolja, hogy az itt következő tárolt eljárás nem felel meg a valóság követelményeinek, annak sajnálattal bevallom, hogy igen, nem. Nem felel meg. De még ez a leegyszerűsített eljárás is olyan hosszú lesz, hogy kivételesen soronként fogom kommentálni, elmagyarázni, hogy mi is történik pontosan.

Elsőként megtervezzük és létrehozzuk azt a táblát, amelyben vezetni fogjuk, hogy kinek, melyik számláján, melyik időszakra számoltunk ki kamatot. Ezzel nagyjából össze is foglaltam, milyen mezők kellenek bele. A CustomerID például nem, mert a számlák egyértelműen megmutatják, kihez tartoznak. Az AccountID igen, valamint egy tól–ig dátumhatár. valahogy így:

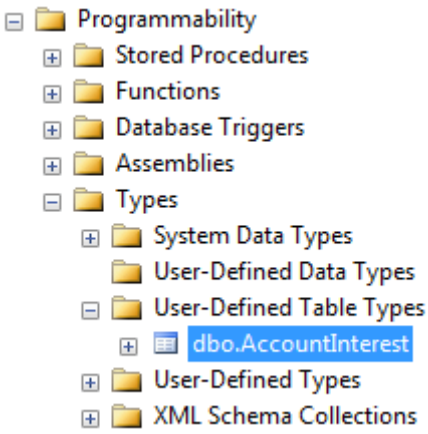
```
CREATE TABLE AccountInterestLog
(
AccountInterestID INT PRIMARY KEY IDENTITY NOT NULL,
AccountID INT FOREIGN KEY REFERENCES Accounts(AccountID),
PeriodStart DATE NOT NULL,
PeriodEnd DATE NOT NULL
)
```

Ezt követően jöhet a tárolt eljárás, melyet felkészítünk tömeges kamatszámításra. Ennek az lesz a módja, hogy az a bemeneti paramétere, amelyik meghatározza, kiken számítson kamatot, tábla-típusú lesz, így egynél több elemet, sőt rengeteget is tartalmazhat. A táblatípus azt is lehetővé teszi, hogy egyedi kamatfeltételeket állapítsunk meg minden egyes ügyfélnek, hisz a bemeneti táblának lehet több oszlopa. Ezzel a lépéssel „outsource-oljuk” a kamatszintet, állítsa be a hívó, hisz egy bankban általában sok tízezer paramétert vesznek figyelembe az adott számla, adott ügyfél, adott lekötés aktuális kamatszintjének meghatározásánál.

Tárolt eljárásnál sajnos nem lehet röptében megadni a táblatípust, hanem először létre kell azt hoznunk a CREATE TYPE utasítással, így:

```
CREATE TYPE AccountInterest AS TABLE(AccountID INT, InterestRate MONEY)
```

Ritka eset, amikor saját adattípussal ajándékozzuk meg magunkat, és lám, e komoly probléma kapcsán pont belefutottunk egy ilyenbe. Itt csücsül a táblatípusunk:



51. ábra - tábla adattípus létrehozása

Ezenfelül megadunk két dátumparamétert, ami azt hivatott ellenőrizni, hogy kétszer ne lépjünk ugyanabba a folyóba, amin már elszámoltunk kamatot, azon most ne tegyük. Összességében így fest a tárolt eljárás fejléce:

```
CREATE PROCEDURE InterestCalculator
     @Interests AccountInterest READONLY,
     @PeriodStart Date, @PeriodEnd Date
AS
```

Itt az eddigiekhez képest a READONLY szócska az egyetlen ismeretlen dolog, azt sem kell megérteni, csak be kell írni, ha táblaértékű bemeneti paraméterünk van, enélkül ugyanis hibaüzenetet kapunk. Kötelező orvosság.

Az első utasításunk mindjárt egy erőteljes BEGIN TRAN legyen, hogy bármi, amit csinálunk, ACID legyen, vagy egy az egyben történjen meg, vagy egyáltalán ne!

```
BEGIN TRAN
```

Ezt követően előállíthatunk egy lepedőt a bejövő paraméterek által meghatározott számlákon, melyben már a kiszámított kamat is benne van. Ez egy jó nagy, négytáblás lekérdezés lesz, mert szükségünk lesz az Accountsra, a Transactionsra, a bemenő táblaváltozóra, valamint az imént létrehozott naplótáblára. Ez utóbbira oly módon, hogy akkor van találat, ha nincs találat, vagyis még nem csináltuk meg az adott kamatszámítást *(OUTER JOIN)*. A lepedőt megvalósító lekérdezés így néz ki:

```
SELECT a.AccountID, Currency, SUM(Amount * i.InterestRate) as Interest
FROM Accounts a JOIN Transactions t ON a.AccountID=t.AccountID
JOIN @Interests i ON a.AccountID=i.AccountID
LEFT OUTER JOIN AccountInterestLog al ON a.AccountID=al.AccountID
AND (a.CreationDate < @PeriodStart OR a.CreationDate > @PeriodEnd)
WHERE al.AccountID IS NULL
GROUP BY a.AccountID, Currency
```

Trükkös! Figyeljük meg, hogy a dátumhatárokat nem a WHERE-feltételben adtam meg, hanem az utolsó, LEFT OUTER JOIN-hoz írtam hozzá! Ennek következtében a dátumválasztás is a JOIN-feltétel

része lett, így ezekre is hat az OUTER JOIN, noha a vizsgált változók nem is a táblákban vannak, hanem bemeneti paraméterek (@PeriodStart és @PeriodEnd). Ezzel a trükkel értem el, hogy az OUTER JOIN legenerálja azokat a sorokat, amihez nincs AccountInterestLog az adott dátumhatárokon belül.

Hogyan készítsünk sorokat nemlétező adatok alapján? OUTER JOIN-nal! O.J. a nagy nullgenerátor!

Végül nekünk pont azok a sorok kellenek, ahol az AccountInterestLog tábla csupa NULL, a többit úgy kivágom a WHERE-résznél, mintha ott sem lett volna.

Ha valakinek ez a trükkös JOIN megfeküdte a gyomrát, ne búsuljon, valószínűleg SQL-es pályafutásom első tíz évében nem tudtam volna egy lépésben leírni ezt a lekérdezést.

Emellett egy csúnya egyszerűsítéssel az összes tranzakcióra *(tartozik, követel – édes mindegy)* pénznemenként egy GROUP BY segítségével hibásan kiszámítjuk az adott időszakra vonatkozó kamatot. A hiba az, hogy nem vesszük figyelembe, az adott perióduson belül mikor került oda az a pénz, továbbá az aktuális egyenleggel sem törődünk, mert még nem tartunk a triggereknél. A mi bankunkban nem kapsz kamatot azért, ha ott tárolod a pénzedet, ellenben busás, félreszámított kamatot kapsz minden tranzakcióért, amelyet a perióduson belül bármikor végzel. Reklám: Helyezze el betétjét nálunk a kamatszámítást megelőző másodpercben, majd vegye ki egy perccel később, és mi busásan megjutalmazzuk ezért!

Ha valaki azt követeli, csináljuk már meg teljesen hibátlanul, az egyben azt is követeli, hogy legyen ez a könyv dupla ilyen vastag, a példák pedig legyenek ötször ekkorák, oldalakon keresztül csak kód, kód és kód. Nem így lesz. A hibáinkkal együtt fogunk élni. Tudunk róluk.

A lepedő alapján kiszámított értékekre többször is szükségünk lesz, mivel egyrészt jóváírjuk a kamatokat az ügyfeleknél, másrészt naplózzuk, hogy ki kapott kamatot, így az adatokat beleirányítjuk egy táblaváltozóba, hogy meglegyen mindkét lépéshez.

```
DECLARE @t TABLE(
      AccountID INT,
      Currency CHAR(3),
      Interest MONEY
)
```

Mivel ez már „helyesen” tartalmazza az elkönyvelt kamatot, most egy mozdulattal be lehet az egészet szúrni a Transactions táblába.

```
INSERT Transactions(AccountID, Currency, Amount, Partner)
SELECT *, 'Időszakos kamat' FROM @t
```

A naplózás már trükkösebb, de igazából ott sem kell sok mindent tenni, csak az elmentett átmeneti táblából kiemelni DISTICNT-tel az ügyfélsorokat, majd a bemeneti paraméterben megadott két dátummal lementeni.

```
INSERT AccountInterestLog(AccountID, PeriodStart, PeriodEnd)
SELECT DISTINCT AccountID, @PeriodStart, @PeriodEnd from @t
```

Nagyjából készen is vagyunk, még kell egy COMMIT.

```
COMMIT
```

Hogy meglegyen egyben is, itt a teljes tárol eljárás mindenenestül:

```
CREATE PROCEDURE InterestCalculator
      @Interests AccountInterest READONLY,
      @PeriodStart Date, @PeriodEnd Date
AS
BEGIN TRAN
DECLARE @t TABLE(
      AccountID INT,
      Currency CHAR(3),
      Interest MONEY
)

INSERT @t
SELECT a.AccountID, Currency, SUM(Amount * i.InterestRate) as
Interest
FROM Accounts a JOIN Transactions t ON a.AccountID=t.AccountID
JOIN @Interests i ON a.AccountID=i.AccountID
LEFT OUTER JOIN AccountInterestLog al ON a.AccountID=al.AccountID
AND (a.CreationDate < @PeriodStart OR a.CreationDate > @PeriodEnd)
WHERE al.AccountID IS NULL
GROUP BY a.AccountID, Currency

INSERT Transactions(AccountID, Currency, Amount, Partner)
SELECT *, 'Időszakos kamat' FROM @t

INSERT AccountInterestLog(AccountID, PeriodStart, PeriodEnd)
SELECT DISTINCT AccountID, @PeriodStart, @PeriodEnd from @t

COMMIT
```

Ha belegondolunk, hogy minden műveletet simán befedtünk egy-egy nem túl bonyolult SQL-utasítással, azt kell mondjuk, az SQL-nyelvnél nincs hatékonyabb a világon. Szinte alig írtunk kódot. Szőröstül-bőröstül, számításostul, hibakezelésestül, naplózásostul mindent meg tudtunk csinálni 8-10 utasítással!

A hatodik hangszóró azé, aki megmondja, miért rossz ez az egész úgy, ahogy van, természetesen az eddig beismert hibákon felül.

Várok…!

Senki?

Akkor elmondom. Úgyse lett volna jó az a hatodik hangszóró semmire sem. Többfelhasználós környezetben simán el tudunk indítani egymással párhuzamosan ugyanarra a periódusra, ugyanazokra a bankszámlákra szinte akárhány kamatszámítást, és mindegyik önállóan, autonóm

módon szépen kiszámítja, hozzáadja a kamatot, majd tisztességesen naplóz is. Mi meg csak nézünk ki a fejünkből, hogyan futhatott ez le ötször. Hát kérem úgy, hogy – egyszerre. Nem mindig öröm, ha valami önmagával párhuzamosan, sokszorosan le tud futni.

Az a lépés a hibás, amikor kiválasztjuk a nem naplózott Accountokat, mert ezt akárhány párhuzamos lekérdezés meg tudja tenni. Ez olyan eljárás, amelyiknél elejét kellene venni annak, hogy önmagával párhuzamosan fusson. Erre egy OPTIMIZER HINT-et vethetünk be. Egy jól irányzott kizárólagos zárral (Exclusive Lock) elejét vehetjük a tárolt eljárásunk párhuzamos futtatásának. Soros feldolgozásúvá tehetjük a tárolt eljárásunkat, ha az AccountInterestLog táblanév mögé beírjuk, hogy

```
LEFT OUTER JOIN AccountInterestLog al WITH (TABLOCKX) ON
a.AccountID=al.AccountID
```

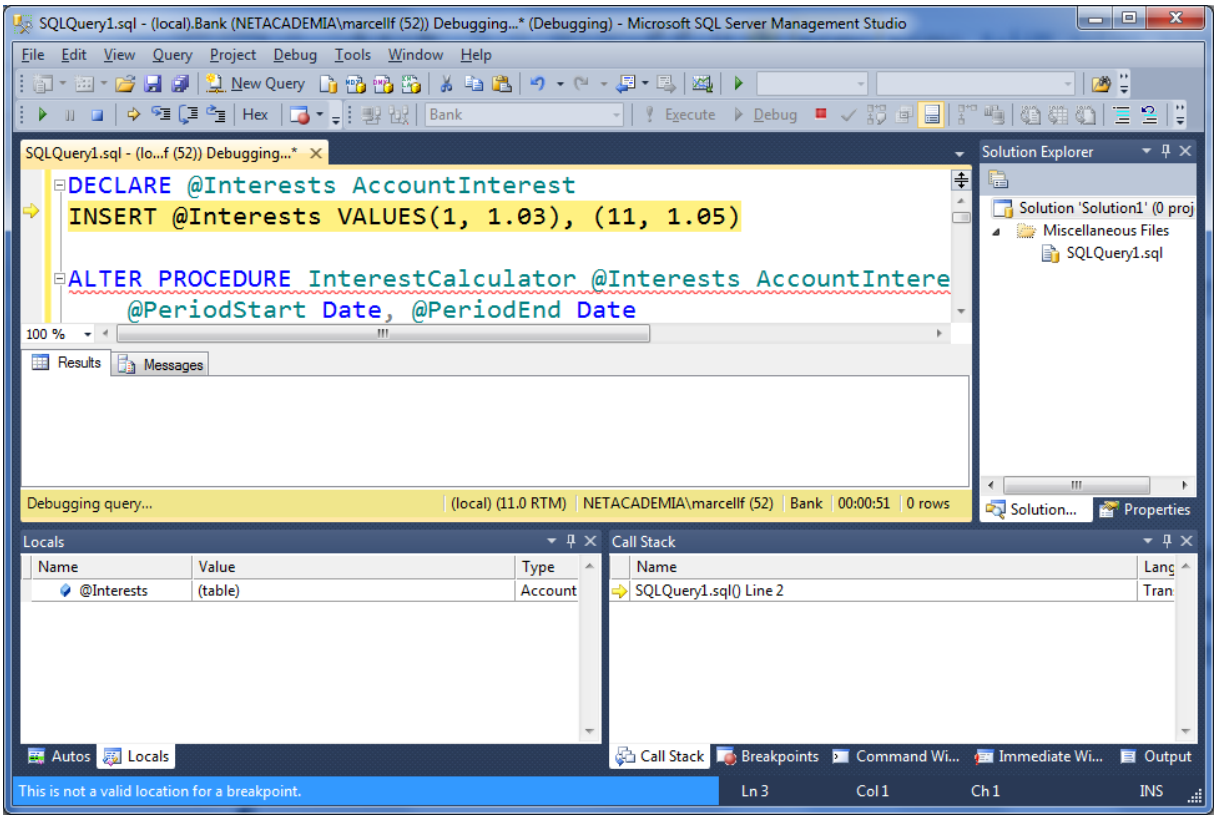
De ez már nagyon haladó dolog, inkább nem is mondtam, nehogy valaki mostantól szíre-szóra begépelgesse ezt. Csak ésszel!

A tárolt eljárásról is meg kell említeni, hogy neki is van WITH ENCRYPTION lehetősége, a nézetekhez hasonlóan – és hasonló problémákkal (többé nem scriptelhető, nem sémakomparálható stb.).

Ha valakinek van egy nagyobbacska tárolt eljárása, most már jól jöhet, hogy az SSMS teljes értékű beépített hibakeresőt, Debuggert tartalmaz. A fenti tárolt eljárás meghívásához az alábbi kódra van szükség, melyet kijelölve a Debug gomb megnyomásával lépésenként futtathatunk:

```
DECLARE @Interests AccountInterest
INSERT @Interests VALUES(1, 1.03), (11, 1.05)
```

Van itt minden, lépésenkénti futtatás, helyi változók, hívási verem (Call Stack), mint a nagyoknál:



52. ábra - lépésenkénti futtatás a Debuggerben

10.5 Trigger

A tárolt eljárások egy másik típusa a trigger, melyet –a fenti példától eltérően – nem mi hívunk meg, hanem az SQL Server. Triggert ugyanis bizonyos eseményekre rakhatunk, amelyek az esemény bekövetkeztekor automatikusan lefutnak.

Ha automatikusan futnak le, abból az következik, hogy a triggereknek nincs bemenő paraméterük, mert nincs, aki kitöltse azokat. És mivel a háttérben, rejtetten futnak, nem igazán jó ötlet adatokat visszaadniuk futás közben (pl. SELECT, RETURN), mert erre nem számít senki.

Triggert egyrészt tranzakciós táblaműveletekre lehet illeszteni (INSERT, UPDATE, DELETE), másrészt az SQL Serverben szerverszinten elképesztően sok mindenre, loginok létrehozására, tábla módosítására és gyakorlatilag mindenre, ami mozog. Ez utóbbiakat hívjuk DDL-triggereknek, és ráhagyjuk ezeket a rendszergazdákra. Mi a táblákhoz rendelt triggereket fogjuk eszményi céljainkhoz felhasználni.

Mielőtt leírnánk az első utasítást, még egy kis magyarázat. Táblára biggyeszthető triggerből kétféle van, amely az alábbi három:

- INSTEAD OF: helyettesítő trigger, egy művelet eltérítésére, kiváltására való. Ha például készítünk egy INSTEAD OF triggert egy törlési műveletre, többé nem a gyári, eredeti DELETE-utasítás fog lefutni törlési művelet esetén, hanem az általunk írt helyettesítőkód, amivel például tényleges törlés helyett logikai törlést valósíthatunk meg, egy mezőben átbillentve a sor töröltségének állapotát, vagy teljesen ki is iktathatjuk az adott műveletet.
- AFTER: az adatmódosító művelet után(?) lefutó kód. A neve teljesen félrevezető, mert nem a tranzakció után fut le, hanem a kellős közepén, ezáltal lehetőségünk van a módosítási adatok figyelembevételével például korrigálni az adatokat, vagy éppenséggel visszavonni a tranzakciót (ROLLBACK). Miféle utána-trigger az, ami a műveletet meg nem történtté tudja tenni? Semmiféle. Ennek ez a neve, de csak azért, hogy jól összezavarja a kezdő SQL-programozót.
- FOR: egy adott műveletre vonatkozó trigger. Na kérem, ez ugyanaz, mint az AFTER, a két kulcsszó csereszabatos. Hogy miért? Ennek történelmi okai vannak, észérvek viszont nincsenek. Ezért írhattam fentebb, hogy ez a kettő az alábbi három…

Ha egy triggernek se bemeneti, se kimeneti értéke nincs, az nem olyan, mint a süketnéma vak? Majdnem, de nem teljesen, mivel más csatornákon mégiscsak hozzájutnak bemenő adatokhoz. A táblaműveletekhez rendelt triggerek *(előtte, utána, helyette, mindegy)* hozzáférnek az adott művelethez tartozó adatokhoz, mégpedig a már megismert INSERTED, DELETED pszeudotáblák segítségével. Vajon mit lát egy INSERT trigger az INSERTED táblában? Úgy van, a beszúrásra váró sorokat. És mit lát a DELETE trigger a DELETED táblában? No, egy hangszóróért? És végül az UPDATE, amely a DELETED-ben látja a múltat, az INSERTED-ben pedig a jövőt. Ez legalább következetes.

Első feladatunk az egyes bankszámlákhoz tartozó egyenleg kiszámítása és folyamatos karbantartása lesz az Amount táblában. Ne feledjük, hogy az egyes tranzakciókból kiszámított egyenleg redundáns adat, amit ha nem tartunk karban, hamis értéket fog mutatni[21]! Ehhez elsőként meg kell alkotni az Accounts táblában egy Balance nevű mezőt az összegek tárolására:

```
ALTER TABLE Accounts ADD Balance MONEY NOT NULL DEFAULT 0
```

Amint ez az utasítás lefut, máris van egy teljesen rossz mezőnk, amely minden számlára nulla egyenleget mutat, úgyhogy ki kell számolnunk az egyenlegeket. De vigyázat, onnantól állandóan karban kell tartani az egyenleget, valahányszor új tranzakció keletkezik, mert ha nem indul be rögtön a karbantartás, az egyenlegek könnyen szétcsúszhatnak ismét. Ezért előbb beüzemeljük a folyamatos karbantartást végző triggert, és csak azt követően korrigáljuk a nullákat az egyenlegmezőben.

Találjuk ki, mely műveletekre kell triggert akasztanunk! INSERT? UPDATE? DELETE? Mivel itt banki tranzakciókról van szó, gyakorlatilag kizárt, hogy egy ilyen sor utólag módosuljon, és persze törlés sem fordulhat elő. Jobban mondva most még igen, de később meg fogjuk oldani, hogy ne fordulhasson elő más, mint INSERT. Ennek következtében egy sima INSERT triggerre van szükségünk, az AFTER (=FOR) fajtából, mégpedig a Transactions táblán. Valahányszor beesik egy új pénzügyi tranzakció, mi felnyúlunk az Accounts táblába, és korrigáljuk a Balance mezőt. Meg lehet ezt oldani egy utasítással? De meg ám!

[21] A mi példánkban így is, úgy is hamisat fog mutatni, mert most szépen elhanyagoljuk a pénznemeket, és lazán összeadjuk az eurót a forinttal … merthogy a korrekt megvalósítás további táblákat, további kódot igényelne.

```
CREATE TRIGGER Balance ON Transactions
FOR INSERT
AS
UPDATE Accounts SET Balance=Balance+Amount
FROM Accounts a JOIN INSERTED i on a.AccountID=i.AccountID
```

A triggerben egy UPDATE utasítás csücsül, ami az Amount táblát összekapcsolja az INSERTED táblával, és onnan az Amount-okat a megfelelő Balance-ba szórja[22]. Ezzel beüzemeltük a karbantartórutint. Jöhet a sok-sok nulla kijavítása. A meglévő banki tranzakciók alapján készítünk egy összegző lekérdezést, melyet ráborítunk az Accounts táblára:

```
UPDATE Accounts SET Balance=ISNULL((SELECT SUM(Amount)
FROM Transactions WHERE AccountID=a.AccountID), 0)
FROM Accounts a
```

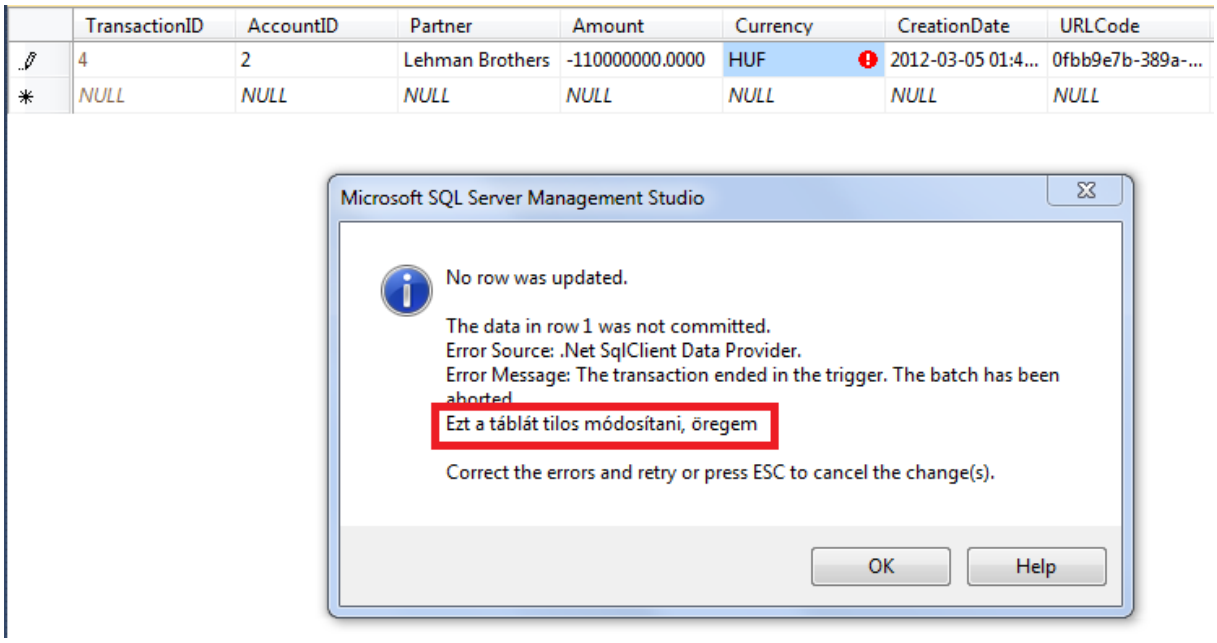
Az ISNULL()-függvényre azért van szükség, mert 1.) az UPDATE nem viseli el a GROUP BY-t, ami kellett volna az összegzéshez, ezért 2.) beágyazott SELECT-et használtam, ami 3.) nem egy INNER JOIN, tehát 4.) nem létező Transactionok esetén is érinti az Account sort, és ott 5.) NULL-t ad vissza, ezt 6.) az ISNULL() visszacseréli nullára. Csak hogy ne szálljon el. ☺

A következő feladat, hogy ne lehessen a Transactions tábla sorait módosítani. Ezt el lehetne érni jogosultság-beállításokkal is, de mi most triggerversenyben vagyunk, ezért triggerrel fogjuk megoldani, mégpedig úgy, hogy ha bejön egy UPDATE, mi visszaborítjuk a tranzakciót:

```
CREATE TRIGGER NoUpdateTran ON Transactions
FOR UPDATE
AS
PRINT 'Ezt a táblát tilos módosítani, öregem'
ROLLBACK TRAN
```

Ez a trigger állhatna akár egyetlen utasításból is, de mi jólneveltek vagyunk, ezért írunk valamit a kimenetre, mielőtt lecsapnánk a tranzakciót, mint a taxiórát.

[22] Talán már megszokhattuk, de ez a trigger is hibás. *(Az egyszerűsítésnek ára van.)* Ha valaki az T. Olvasók közül meg tudja mondani, mi rossz rajta, nem egy hangszórót, de egy Túró Rudit kap ajándékba!



53. ábra - munkában a triggerünk

Most akadályozzuk meg a törlést is! Ezúttal használjunk INSTEAD OF triggert, ami az eredeti DELETE művelet helyett az általunk megadott kódot futtatja le, ami lehet például egy naplózás, hogy ki, mikor kísérelte meg a törlést! Maga a törlés azonban nem fut le:

```
CREATE TRIGGER NoDeleteTran ON Transactions
INSTEAD OF DELETE
AS
INSERT LogTable(Username, CreationDate)
     VALUES(SUSER_SNAME(), GETUTCDATE())
```

Ez a trigger a LogTable nevű (nem létező) táblába írja a törlési kísérleteket (tehát elszáll ☺) – de nem töröl.

Vegyük észre, hogy a jogosultságokkal szemben most mindenki számára megszüntettük a módosítás, törlés lehetőségét, Adminka sem tud törölni, módosítani a triggerek lelövése nélkül, ami viszont nagy kényelmetlenséggel és rizikóval jár, tehát nem fogja őket kikapcsolni.

10.6 Skaláris és táblafüggvények

A harmadik tárolt eljárásszerűség a függvény. Ez a konstrukció az SQL Server 2005-ös verziójában jelent meg, bátran állíthatom, hogy sokéves felhasználói hiszti után. Én magam is hisztiztem érte, onnan tudom. ☺ Az indulatokat az váltotta ki, hogy a tárolt eljárások bár nagyon jól paraméterezhetők és programozhatók, felhasználási körük azonban szűkös. Egy tárolt eljárást nem lehet mezőlistában használni, táblaforrásként megadni, WHERE-feltételben szerepeltetni. A tárolt eljárást futtatni lehet, és ezzel *(majdnem[23])* vége. Ha tehát valaki kemény munkával bevisz valami üzleti logikát (például kamatos kamat) egy tárolt eljárásba, azt utána kenheti a hajára, mert SELECT-

[23] Az egyetlen lehetőség a futtatáson kívül, hogy a tárolt eljárás eredményhalmazát bele lehet lökni egy INSERT-utasításba.

ben nem fogja tudni felhasználni. Ezt a hiányosságot pótolják a függvények, amelyekből 2005-ben egyszerre kaptunk vagy negyedtucatot!

Az első függvénytípus a skaláris függvény, ami nagyjából annyit jelent, hogy a függvény egy darab értéket ad vissza. Az egyértékű függvények felhasználása a legszélesebb körű, mert mindenhol szerepelhetnek, ahol egy SQL-utasításban kifejezés állhat: mezőlista, WHERE-feltétel, HAVING stb.

A második típus az egysoros táblafüggvény, ami leginkább egy paraméterezhető nézet. Milyen jó is lenne, ha a nézeteinknek bemenő paramétert adhatnánk! Adhatunk, csak akkor az már függvény. Egysorosnak azért nevezzük, mert a nézetekhez hasonlóan egy darab SELECT lehet bennük és kész. Persze a bemeneti paramétereket ez a lekérdezés felhasználhatja. Az egysoros táblafüggvény mindenhol szerepelhet, ahol tábla szerepelhet, tehát FROM záradékban, virtuális táblaként!

A harmadik típus pedig az ereszd-el-a-hajamat táblafüggvény, amiben tetszőleges bonyolultságú rutin akár soronként pakol össze egy táblát, és a végén boldogan visszatér. Ennek eredménye is egy táblaként felhasználható valamicsoda.

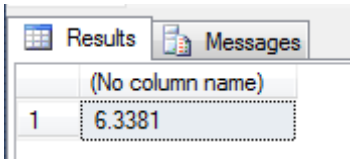
Kezdjük az egyszerűbbekkel, és onnan haladjunk a bonyolultabbak felé! Készítsük el a kamatos kamat függvényt, mely egy kamatráta és egy futamidő segítségével kiszámítja egy bemenő összeg x év múlva esedékes értékét! Maga a kamatos kamat borzasztó egyszerű dolog, az alapösszeget meg kell szorozni a kamat annyiadik hatványával, ahány évnyi kamatozásról beszélünk. Lássuk a függvényt!

```
CREATE FUNCTION KamatosKamat(@Alap MONEY, @Kamat MONEY, @Ev FLOAT)
RETURNS MONEY
AS
BEGIN
      RETURN @Alap * POWER(@Kamat, @Ev)
END
```

Itt már kötöttebb a szintaxis, mint a tárolt eljárásoknál, ami nem is csoda, hiszen vagy tíz évvel később született. Nem lehet eltrehánykodni például a függvény elejét és végét (BEGIN, END), amit a tárolt eljárásoknál még lazán elhagyhattunk. A paraméterek zárójelben vannak, ahogy ez egy rendes függvénytől elvárható, és meg kell adni a visszatérési érték adattípusát is (RETURNS kulcsszó). A többi – remélem – önmagyarázó. Újdonsült függvényünket akár tört évekkel is használhatjuk:

```
SELECT dbo.KamatosKamat(3, 1.12, 6.6)
```

A függvény neve elé a vasszigor nevében ki kell írni a sémáját (jelen esetben: dbo). És az eredmény:



Hat és fél év múlva a három petákunkból 12%-os éves kamattal számolva 6,33 peták lesz.

Most tegyük bele a függvényt egy lekérdezésbe, kérdezzük le a kuncsaftjaink egyenlegét 6%-os kamattal, tíz évre!

```
SELECT AccountID, dbo.KamatosKamat(Balance, 1.10, 10)
```

```
FROM Accounts
```

Ezt tudják a skaláris függvények.

Egysoros táblafüggvényként pedig készítsünk egy függvényt, ami egy bemenő paraméter mentén válogatja le a tranzakciókat! Ez most nem annyira jó példa, mert ezt egy nézettel is simán meg lehet csinálni, de a bonyolultabb táblafüggvény felé vezető úton egy szükséges lépés.

```
CREATE FUNCTION EgysorosTabla(@Kuszob MONEY)
RETURNS TABLE
AS
RETURN
      SELECT AccountID, Balance FROM Accounts
            WHERE Balance > @Kuszob
```

Ebben az egyszerű esetben a függvény törzsét alkotó SELECT határozza meg a visszaadott tábla szerkezetét, ezért a tábladefinícióval nem kell külön foglalkoznunk. A bemeneti paramétert a WHERE-feltételben látjuk viszont. Függvényünk táblaértékű, tehát FROM záradékban szerepelhet táblaként, így:

```
SELECT * FROM dbo.EgysorosTabla(4)
```

Egyszerű, mint a faék. Az összett táblafüggvény, melyben kézzel, menet közben állítunk össze egy táblát, már más tészta. Készítsünk egy olyan táblaértékű függvényt, ami a mai naptól számítva megadott darabszámú „hóelejét”, azaz a következő hónapok első napjait adja vissza, mert szükségünk van egy olyan táblára, amit más táblákhoz tudunk kapcsolni, és „hóelejék” vannak benne. Ha én volnék én, az ilyen táblát bizony egy ciklussal állítanám össze. És az már többsoros kód, akárhogyan is nézzük.

```
CREATE FUNCTION Hoeleje(@Honapok INT)
RETURNS @t TABLE(FirstDay DATETIME) --tábladefiníció
AS
BEGIN
      WHILE @Honapok > 0
      BEGIN
            INSERT @t VALUES (DATEADD(DAY, 1, DATEADD(MONTH, @Honapok-
            1, EOMONTH(GETDATE()))))
            SET @Honapok-=1 --ciklusváltozó csökkentése
      END
      RETURN --a kész tábla visszaadása
END
```

Akinek a dátumszámítós rész nem működik, az ismét lebukott, hogy nem SQL 2012-t használ, mert az EOMONTH() függvény 2012-es újdonság, és ahogy a neve is sugallja, egy dátum alapján előállítja a hónap utolsó napját. Ezzel túl egyszerű lett volna a feladat, ezért döntöttem a hónap első napja mellett. ☺

Érdemes megfigyelni, hogy semennyit sem törődtem a legenerált „hóelejék” sorrendjével (konkrétan „fejjel lefelé” generálom őket), hisz a végeredményünk egy tábla, amit majd a felhasználója úgy rendez sorba, ahogy akar:

```
SELECT * FROM Hoeleje(9) ORDER BY 1
```

Ha valaki szemfüles volt, kiszúrhatta, hogy nem készítettünk saját aggregátumfüggvényt. Egyik függvényünk sem használható GROUP BY kifejezésben a SUM(), AVG() és társaikhoz hasonlóan. Nem véletlenül. Aggregátumfüggvények nem készíthetők Transact SQL-ben, mert azok két fázisból állnak, egyrészt egy soronkénti végrehajtásból (ez oké, ez megvan), másrészt egy részösszegképző rutinból, mint amilyen az AVG() függvénynél az osztási művelet. Ez utóbbi létrehozására nincs SQL szintaxis. Ha valakinek szüksége lenne egy alternatív SUM()-ra, mely figyelembe veszi az ügyfél bónusz-málusz-vip akármijét, azt is el lehet készíteni - .net-ben. Az sem ördöngösség, de kimutat az SQL-világán kívülre, ezért nem foglalkozom vele.

Ezzel áttekintettük az SQL Server programozási lehetőségeit, jöhetnek a különleges adattípusok!

11 Spéci adattípusok

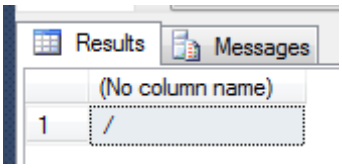
Az SQL Serverben van egy-két olyan adattípus, aminek a puszta látványától feláll a kezdők hátán a szőr. Ilyen az XML, a HierarchyID, a Geometry-Geography és általában azok az adattípusok, amelyek plusz funkciókkal rendelkeznek. Az INT, a DATETIME, a FLOAT és a többiek buta adattípusok, önmagukban nem tudnak semmit. Egy XML-adattípusnak azonban nemcsak lelke, de metódusai vannak, amelyekkel objektumorientált módon lehet és kell zsonglőrködnünk. Nemcsak értékük van, mint egy INT-nek, hanem programlogika is tartozik hozzájuk.

Miért ilyen faramuci adattípusok ezek? Azért, mert ezek mind .net-ben vannak megvalósítva (külső DLL-ek), vagyis minden egyes ilyen adattípus egy-egy dotnet-osztály példánya. És mivel az, ami, tartoznak hozzá dotnetes propertik és metódusok. Ha dotnetesek, akkor például mindegyiknek van ToString() metódusa, ami a benne rejlő bináris adatot stringformátumban vissza tudja adni nekünk. Sőt, a dolog fordítva is működik, mert ezeknek a jószágoknak van Parse() metódusuk, tehát ha stringet tömködünk beléjük, fel fogják dolgozni!

Különösebb előmagyarázat nélkül nézzük például ezt a kis aranyos HierarchyID-t, amibe ostobán beletöltünk egy nulla értéket (hexa nulla kell neki, nem INT, ezért a 0x), majd visszakérjük string formátumban:

```
DECLARE @akarmi HierarchyID=0x
SELECT @akarmi.ToString()
```

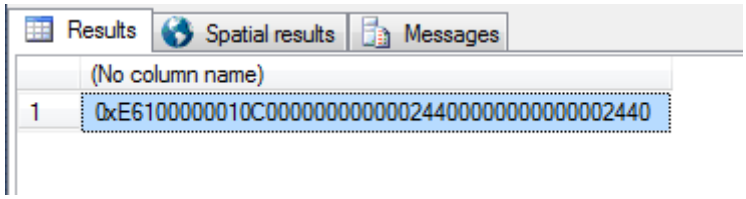
És amit kapunk az egy per jel:



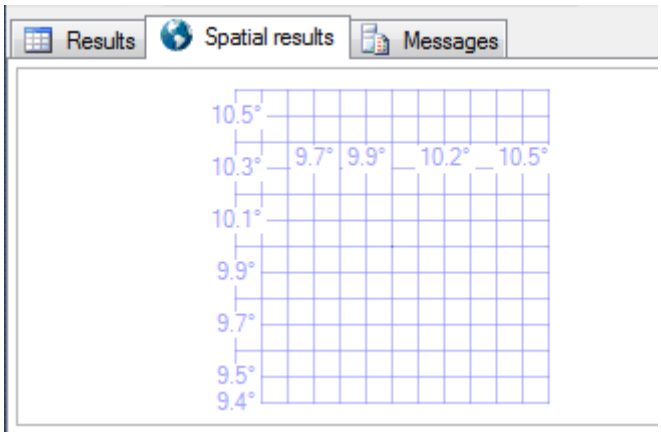
Ez biztató, ennek valami köze lehet az útvonalakhoz! Most fordítsuk meg, vegyünk egy másik dotnetes adattípust, és tömjünk bele stringet, mire megyünk vele?

```
DECLARE @akarmi GEOGRAPHY='POINT(10 10)'
SELECT @akarmi
```

Az eredmény „önmagáért beszél”. ☺



Ami a lényeges, hogy valami biztosan történt, mert lett egy olyan fülünk, hogy Spatial results, ahol ugyan a pontunk nem látszik, mert túl pici, de a koordináták valamiféle 10, 10 bigyó közelében állnak, tehát beletrafáltunk!



Megállapodhatunk tehát abban, hogy a dotnetes adattípusok ugyan a mágia területére tartoznak, de kapcsolatot tudunk velük teremteni, ha másképp nem, hát füstjelekkel és stringüzenetekkel. Remélem, ettől mindenkinek elszállt a félelme és/vagy undora, megjött a bátorsága, úgyhogy megvizsgálhatjuk ezeket a csodabogarakat egyesével, részletesen is.

11.1 Az XML-adattípus

XML-formátumra általában akkor van szükségünk, ha az adatainkat külső rendszer számára szeretnénk átadható, elfogadható állapotra hozni. Az SQL Server igen gazdag SQL-formázási lehetőséggel rendelkezik, melynek használata a pofonnál is egyszerűbb. Ha le szeretnénk kérdezni mondjuk a tranzakcióinkat XML-formátumban, egyszerűen a lekérdezés végére kell írni, hogy FOR XML AUTO, és a dolog le van tudva, így:

```
SELECT * FROM Accounts FOR XML AUTO
```

XML-t előállítani tehát nevetségesen egyszerű, még akkor is, ha valakinek nem pont ez az XML-elrendezés tetszik, mert van FOR XML ELEMENTS például, de a legszebb a FOR XML PATH, ami úgy működik, hogy minden mezőt az XML-hierarchia tetszőleges mélységébe generálhatunk pusztán a mező becenevek (aliasok) megadásával, mondjuk így:

```
SELECT     AccountID AS 'alatta',
           CustomerID AS 'alatta/meglejjebb',
           AccountNumber AS 'alatta/mellette',
           Balance AS 'masik/agon/ul/egy/vereb'
           FROM Accounts FOR XML PATH
```

Az eredmény:

```
<row>
  <alatta>1<meglejjebb>1</meglejjebb><mellette>12345678-64353453</mellette></alatta>
  <masik>
    <agon>
      <ul>
        <egy>
          <vereb>0.0000</vereb>
        </egy>
      </ul>
    </agon>
  </masik>
</row>
<row>
  <alatta>2<meglejjebb>2</meglejjebb><mellette>9876543-64353453</mellette></alatta>
  <masik>
    <agon>
      <ul>
        <egy>
          <vereb>-110000000.0000</vereb>
        </egy>
      </ul>
    </agon>
  </masik>
</row>
```

54. ábra - tetszőleges kialakítású XML-struktúra

A FOR XML záradék egyébként még kismillió formázási lehetőséget tartalmaz, a gyökérelem megadásától az XML-sémák használatáig. Ha valakinek ennyire speciális XML-es feladata akad, nézzen utána!

A következő mutatvány egy XML-ben megkapott adathalmaz beolvasása, táblázattá alakítása. Az sem bonyolultabb, csak tudni kell a nyitját, ami nem más, mint egy táblaforrásként használható függvény, az OPENXML. Ennek van egy kezdőknek szánt változata, ami úgy működik, hogy az XML-dokumentumban egy adott SQL-tábla mezőneveit automatikusan megkeresi, és az értékeket táblázatos formára hozza. Ha tehát az XML-doksiban szerepel AccountID és CustomerID tag, mint az alábbi példában is, az OPENXML automatikusan kiguberálja az értékeket, mivel megadtuk neki, hogy az Accounts tábla szerkezetét keresse az adatokban.

Sajnos a kód áttekinthetetlensége érdekében foglalkoznunk kell azzal a csekélységgel is, hogy az OPENXML nem magát az XML-szöveget, hanem annak feldolgozott, memóriabeli reprezentációját várja bemenetként, amit egy INT típusú kezelőn (Handle) keresztül érünk el. Emiatt van szükség a @docHandle változóra és az sp_xml_preparedocument tárolt eljárásra. Kezdőknek…

```
DECLARE @doksi
XML='<akarmi><AccountNumber>23</AccountNumber><CustomerID>21323</Cus
tomerID></akarmi>'

DECLARE @docHandle INT
EXEC sp_xml_preparedocument @dochandle OUTPUT, @doksi

SELECT * FROM OPENXML(@docHandle, '/akarmi', 2) WITH Accounts
```

A harmadik mutatvány pedig maga az XML-adattípus. Mint a fenti példában látható, lokális változónak, de egyébként tábla mezőjének is adhatjuk ezt a típust, ami innentől lehetővé teszi, hogy ha minden rekordunkhoz tartozik egy XML-doksi, akkor azt a rekordban magában tároljuk, ne kelljen a fájlrendszerben idétlen nevű egyedi fájlok rengetegével kínlódni.

Az XML-adattípus pedig állam az államban, vagy csepp a tengerben, kinek hogy tetszik, mert ő maga ott bent a rekordban egy dokumentum, amiben például keresni lehet. A fenti példát felhasználva lássuk, hogyan lehet egy XML-adattípusból kiszedni egy bizonyos értéket! Hát a Query() metódus felhasználásával!

```
SELECT @doksi.query('/akarmi/AccountNumber')
```

Ez a lekérdezés kiszedi a hivatkozott XML-típusú változóból, vagy mezőből az /akarmi/AccountNumber útvonalon található XML-elemet. Ha pedig nem az elemre, hanem annak értékére lenne szükségünk, a value() metódus lesz a mi barátunk:

```
SELECT @doksi.value('(/akarmi/AccountNumber)[1]', 'INT')
```

A szintaxisát szokni kell – vagy megtanulni.

Az XML-adatok teljes körű kezeléséhez ismét házi feladatot kell adnom, mert a témakör messze túlmutat az SQL-nyelven és az SQL Serveren. Aki ennél bővebben szeretne megismerkedni a lehetőségekkel, olvassa el az XQuery és az XPath nyelvek dokumentációját!

11.2 Főnök-beosztott viszony HierarchyID-vel

A SELF JOIN-nál hoztuk létre az Employees táblát, benne a bank alkalmazottaival, valamint a főnök-beosztott viszonyt leképező referenciális integritási szabállyal: a BossID mező mutatja meg, melyik dolgozónak ki a közvetlen főnöke. Abban a fejezetben nyitva hagytuk a kérdést, hogy ha szükségünk lenne egy vezető összes beosztottjára tetszőleges mélységben, azt hogyan kérdeznénk le.

Ha elszakadunk a hagyományos hierarchiaábrázolástól és áttérünk a HierarchyID adattípus használatára, ez a feladat pofonegyszerűen, egyetlen lekérdezéssel végrehajthatóvá válik, mivel a HierarchyID meg tudja adni minden egyes személy „útvonalát” a hierarchiában. Onnantól a beosztott beosztottjának keresése leegyszerűsödik arra, hogy kérem mindazokat, akiknek az útvonala úgy kezdődik, mint a főnök útvonala.

Valósítsuk ezt meg a gyakorlatban!

Elsőként hozzáadunk két mezőt az Employees táblához. Az első egy HierarchyID típusú mező lesz, a második pedig egy számított mező, amely az adott rekordhoz tartozó útvonalat tartalmazza.

```
ALTER TABLE Employees ADD Hierarchy HierarchyID
ALTER TABLE Employees ADD Path AS Hierarchy.ToString()
```

Az útvonal „kiszámítása” nem áll másból, mint a ToString() metódus meghívásából. Ha most elkezdjük feltölteni a Hierarchy mezőt, az útvonal automatikusan kiszámítódik. Szerencsére a HierarchyID mezőbe közvetlenül beleírhatjuk a Path-okat, abból majd ő előállítja a bináris értéket. Már csak

annyit kell tudni, hogy a hierarchia (és az útvonal) számokból épül fel. Ezen információ birtokában kitölthetjük a hierarchiamezőt, így:

EmployeeID	BossID	Name	Hierarchy	Path
1	*NULL*	Mérges Gyula	/	/
2	1	Szorgos Kata	/1/	/1/
3	1	Pancser Géza	/2/	/2/
4	3	Okos Imre	/2/1/	/2/1/
5	3	Lusta Disznó	/2/2/	/2/2/
NULL	*NULL*	*NULL*	*NULL*	*NULL*

55. ábra - hierarchiaértékek begépelése

Látszólag a Hierarchy és a Path mező között nincs különbség, de ez csak a szerkesztőfelület okosságának köszönhető. Egy sima SELECT ezt mutatja, tehát szükséges a Path mező is:

Results | Messages

	EmployeeID	BossID	Name	Hierarchy	Path
1	1	NULL	Mérges Gyula	0x	/
2	2	1	Szorgos Kata	0x58	/1/
3	3	1	Pancser Géza	0x68	/2/
4	4	3	Okos Imre	0x6AC0	/2/1/
5	5	3	Lusta Disznó	0x6B40	/2/2/

56. ábra - a hierarchiaértékek binárisan

Innentől a BossID mezőtől meg is szabadulhatunk.

A HierarchyID is dotnetes adattípus, tehát neki is van egy sereg metódusa. A fontosabbak:

- GetDescendant() – csinálj nekem gyermeket!
- GetAncestor() – apu!
- GetLevel() – hányadik mélységben vagyok?
- IsDescendantOf() – ez a bizonyos valaki az apám?
- GetReparentedValue – családfa átportolása másik ős alá

Ezekhez a metódusokhoz kiváló példákat találunk az SQL Server dokumentációjában.

11.3 Geometry, Geography és a térképrajzolás

És végül nem lenne teljes ez a könyv, ha nem foglalkoznánk a legmókásabb adattípusokkal, a geometriával és a geográfiával. Mindkettő arra való, hogy térbeli információt ábrázoljunk velük. A kettő közötti különbség abban áll, hogy míg a Geometry nem foglalkozik mértékegységekkel, és számára lapos a Föld, addig a Geography valódi térképkoordinátákkal dolgozik, és tudja, hogy anyabolygónk gömbhöz hasonlatos képződmény.

Ha valaki lakberendezésre vagy raktárterület menedzselésére szeretné használni az SQL Servert, használja a Geometry adattípust! Remekül meg lehet állapítani a segítségével területek méretét,

átfedését, távolságát stb. anélkül, hogy törődnünk kellene a mértékekkel. Hogy miben számolunk? Azt az alkalmazás fejlesztője rendeli hozzá. Méter? Milliméter? Egyre megy.

Ha valaki térképen ábrázolandó dolgokat szeretne kezelni és tárolni, neki a Geometry való. Minél nagyobb és távolabbi objektumokról van szó (például: városok), annál fontosabb, hogy a távolságokat ne síkban, hanem a földgömb felületén értelmezze a rendszer. Így is teszi.

Első alkalmazásunk – elszakadva a banktól – egy raktári alkalmazás lesz. Beszórunk a raktárba különböző alakzatokat, aztán megnézzük, befér-e oda még egy zongora. Na jó, ezt nem nézzük meg, mert rémisztően sok kódot igényelne, de más érdekességeket igen. Hozzunk létre egy Raktar nevű táblát, ebbe fognak kerülni a „tárgyak”:

```
CREATE TABLE Raktar(Targy NVARCHAR(50), Helye GEOMETRY)
```

Húzzuk meg a raktár falait:

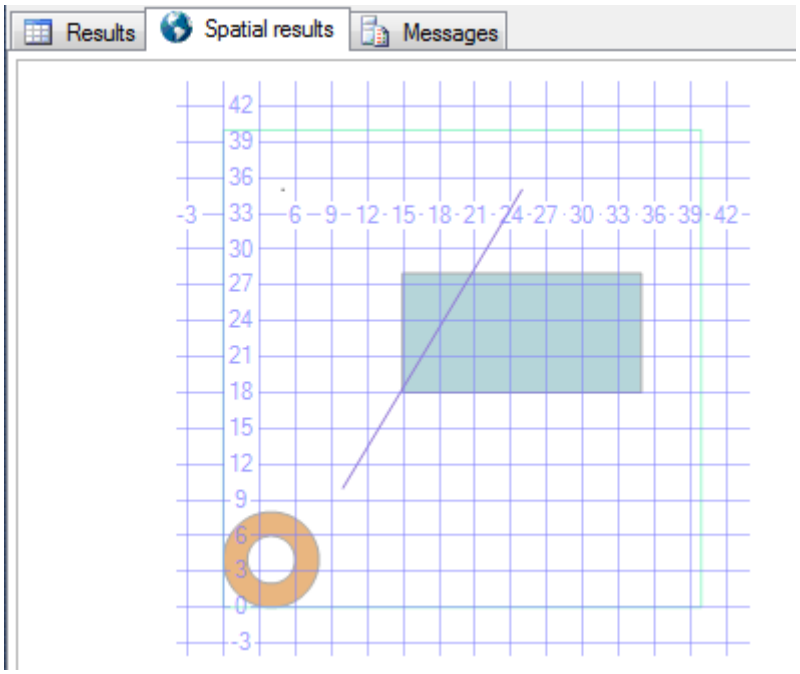
```
INSERT Raktar VALUES('Falak', 'LINESTRING(0 0, 40 0, 40 40, 0 40, 0 0)')
```

Most tegyünk bele a raktárba néhány tárgyat:

```
INSERT Raktar VALUES('Mákszem', 'POINT(5 35)')
INSERT Raktar VALUES('Horgászbot', 'LINESTRING(10 10, 25 35)')
INSERT Raktar VALUES('Asztal', 'POLYGON((15 18, 35 18, 35 28, 15 28,
15 18))')
INSERT Raktar VALUES('Úszógumi', 'CURVEPOLYGON(CIRCULARSTRING(0 4, 4
0, 8 4, 4 8, 0 4), CIRCULARSTRING(2 4, 4 2, 6 4, 4 6, 2 4))')
```

Felhasznált alakzatok: pont (POINT), vonal (LINESTRING), zárt alakzat (POLYGON), görbe (CIRCULARSTRING). Minden zárt alakzat (POLYGON) tartalmazhat lyukakat, így készült az úszógumi is, a külső körből kiveszi a belsőt. Mértékegységet nem értelmezünk, a számok éppúgy jelenthetnek métert, mint rőföt.

A raktárunkban a sarokban van egy úszógumi, keresztben egy pecabot, és még egy asztal is áll a térben. A mákszemről nem is beszélve. Lekérdezve a raktár tartalmát, ezt látjuk:



57. ábra - tárgyak egy raktárban

Mivel mélységi információ nincs, nem tudjuk eldönteni, hogy a horgászbot az asztalon vagy az asztal lába alatt helyezkedik el. Most jön a lényeg! Számítsuk ki a tárgyak által elfoglalt helyet! Dotnetes adattípusról lévén szó, már meg sem lepődünk, hogy metódushívásokkal élhetünk. E két adattípusnál jó, ha megjegyezzük, hogy a metódusok neve ST betűkkel kezdődik, ST, mint standard. A terület kiszámítására az STArea() metódus használható, így:

```
SELECT *, Helye.STArea() AS Terulet FROM Raktar
```

Az alábbi ábrán látható, hogy területtel csak az asztal és az úszógumi rendelkezik, mert ezek zárt alakzatok, poligonok:

	Targy	Helye	Terulet
1	Horgászbot	0x00000000011400000000000024400000000000000244000...	0
2	Mákszem	0x00000000010C0000000000001440000000000000804140	0
3	Asztal	0x000000000104050000000000000000002E400000000000...	200
4	Úszógumi	0x0000000002040A0000000000000000000000000000000000...	37.6991118430775
5	Falak	0x000000000104050000000000000000000000000000000000...	0

A raktárunk területe egyébként 40 x 40, még ha vonallal is vettem körül, így egyszerű kivonással megállapítható, hogy mekkora szabad területünk van. Persze ez csalóka, ha a tárgyak egymás hegyén-hátán vannak, de szerencsére ezt is le lehet kérdezni a STCrosses() metódussal. Nézzük meg, hogy mi metszi az íróasztalt:

```
SELECT *, Helye.STCrosses('POLYGON((15 18, 35 18, 35 28, 15 28, 15
18))') AS Belelog FROM Raktar
```

A horgászbot lóg bele:

Results | Spatial results | Messages

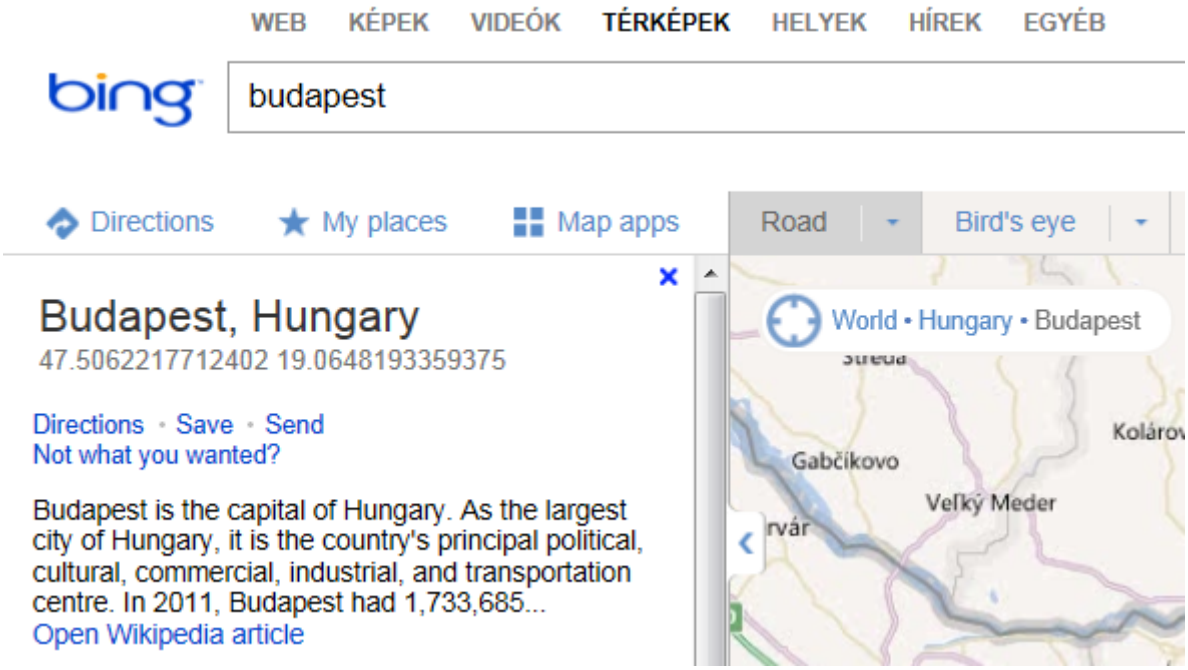
	Targy	Helye	Belelog
1	Horgászbot	0x000000000114000000000000244000000000000244000...	1
2	Mákszem	0x00000000010C000000000000144000000000000804140	0
3	Asztal	0x000000000104050000000000000000002E400000000000...	0
4	Úszógumi	0x0000000002040A000000000000000000000000000000000...	0
5	Falak	0x000000000104050000000000000000000000000000000000...	0

A geometry adattípus metóduslistája több mint 40 elemből áll, ezért ezt nem fejtem ki itt bővebben. Elég annyit tudni, hogy bármilyen kérdésre megadható a válasz a megfelelő metódus meghívásával. Metszi? Közel van? Magában foglalja? stb.

A Geometry és Geography adattípusok édestestvérek. Minden, amit a raktáras példában láttunk, gyakorlatilag módosítás nélkül működik térképészeti esetben is. A különbség a számítások módjában, a gömbfelület figyelembevételében rejlik. A jelenség megismeréséhez kiegészítjük a Cities táblánkat az adott város koordinátáival:

```
ALTER TABLE Cities ADD Location Geography
```

És hogy lássuk, mennyire gömb a gömb, vigyünk fel néhány egymástól távol lévő várost! A városok koordinátáit a Bing Mapsról olvastam le, mert – a Google Maps-szel ellentétben – lelkesen kiírja a bal oldali sávba. Mielőtt betennénk a táblába, a térképről leolvasott koordinátaértékeket fel kell cserélni, mert a Bing mint Microsoft-termék fordítva látja a világot, mint az SQL Server, ami viszont egy Microsoft-termék. Lássunk egy példát, vigyük fel Budapestet!



58. ábra - Budapest földrajzi koordinátái

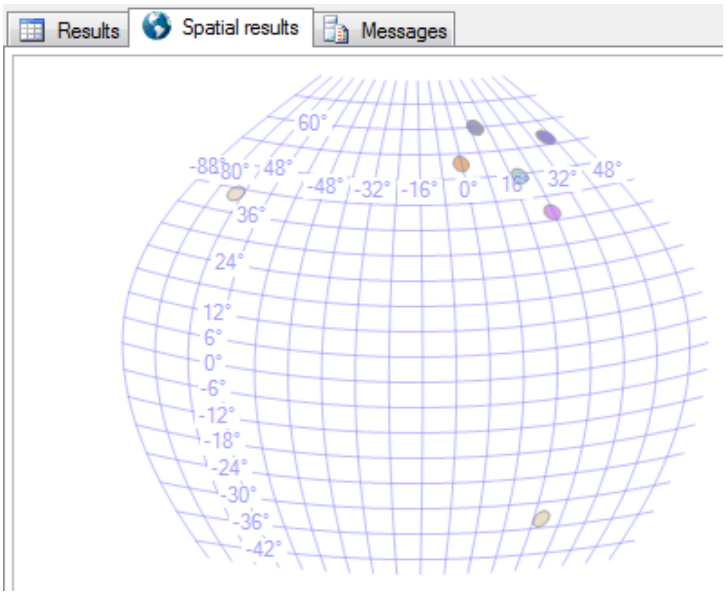
A fentiek szerint tehát ebből így áll elő a megfelelő INSERT-utasítás:

```
INSERT Cities(Name, Location) VALUES('Budapest',
'POINT(19.0648193359375 47.5062217712402)')
```

Még gyors egymásutánban néhány város:

```
INSERT Cities(Name, Location)
VALUES('Budapest', 'POINT(19.0648193359375 47.5062217712402)')
INSERT Cities(Name, Location)
VALUES('London', 'POINT(-0.127140000462532 51.5063209533691)')
INSERT Cities(Name, Location)
VALUES('Moszkva', 'POINT(37.6150093078613 55.7569808959961)')
INSERT Cities(Name, Location)
VALUES('Athén', 'POINT(23.7364101409912 37.9761505126953)')
INSERT Cities(Name, Location)
VALUES('New York', 'POINT(-74.0071182250977 40.7145500183105)')
INSERT Cities(Name, Location)
VALUES('Fokváros', 'POINT(18.421989440918 -33.9190902709961)')
```

Amelyek így festenek a térképen:



59. ábra - világvárosok az SQL Serverben

Két város távolságát a - kitalálható nevű – STDistance() függvénnyel lehet lekérdezni. A térképészeti alkalmazások készítésének csak a fantáziánk szab határt.

12 Az SQL Azure

Az előző fejezetben az SQL Server-rel, annak tulajdonságaival és kezelésével ismerkedhettünk meg. Most a Microsoft SQL Azure-t fogjuk közelebbről szemügyre venni. Az SQL Azure egy felhő alapú relációs adatbázis-kezelő rendszer. Ez egy magas rendelkezésre állású, skálázható, multi-tenant (több-bérlős) szolgáltatás, amit a Microsoft biztosít számunkra a felhő megoldásában. A fejlesztőknek és az üzemeltetőknek mostantól nem kell gondoskodniuk a telepítésről, a beállításokról, patchekről és a menedzselésről, így a fizikai adminisztráció megszűnik, és kizárólag a logikai adminisztráció marad az adatbázis adminisztrátorok számára.

Ha már van egy meglévő adatbázisunk, és azt mi fel szeretnénk tenni a felhőbe, akkor miért ne használnánk a meglévő technológiákat és a tudásunkat? Az SQL Azure ebben segít nekünk, hisz fejlesztési szempontból minimális új tudásra kell szert tennünk ahhoz, hogy a hagyományos SQL Server-es (on-premises[24]) megoldás helyett SQL Azure-t használjunk.

Azoknak a cégeknek, amelyek egy internet alapú szolgáltatást akarnak nyújtani, sokféle kihívással kell szembenézniük. A felhasználók ugyanis az adataikat el szeretnék érni akár többféle készülékről vagy platformról. Az adatbázis méretétől, annak rendelkezésre állásától elég sokféle problémát kell leküzdenünk. Ha a hagyományos modellt tekintjük alapul azaz, hogy a fizikai szerver és vele együtt az SQL Server is a mi fennhatóságunk alatt van, akkor több problémára is figyelnünk kell. Egyfelől szükségünk lesz egy vagy több szerverre, operációs rendszerre, adott termékek licenszére, tárhelyre, hálózatra stb. A rendszergazdáknak figyelniük kell a rendszer állapotát, teljesítményét, rendelkezésre állását, és ha valami probléma van, akkor be kell avatkozniuk. A rendszerünknek természetesen probléma esetén is válaszképesnek kell lennie. Nem engedhetjük meg, hogy egy-egy karbantartás esetén a felhasználók ne érjék el a szolgáltatásokat. Ha belegondolunk abba, hogy egy ilyen típusú rendszer megépítése és üzemeltetése mennyi munkaórába és pénzbe kerül, rájövünk, hogy ez nem egy költséghatékony megoldás. Akkor, hogy lehetne megoldani ezt a problémát költséghatékony módon? Hisz gondoljuk csak bele, mennyibe kerül egy szerver beszerzése, és ha hibatűrő rendszert akarunk, akkor nem egy, hanem több szervert kell vásárolnunk. A licenszek sem olcsók, és még nem beszéltünk a rendszergazdák bérezéséről és a bevezetés költségeiről. Ez így nagyon soknak tűnik. Viszont ha már egy meglévő hibatűrő infrastruktúra, akkor miért ne vásárolhatnánk erőforrást anélkül, hogy pénz hegyeket pazarolnánk el egy saját megoldás kivitelezésére. Ha költséghatékonyabbak szeretnénk lenni ilyen téren, akkor a Microsoft SQL Azure erre kitűnő megoldás! Ráadásul ugyanazokkal az eszközökkel dolgozhatunk, mint eddig, hisz az SQL Azure az eszközeink nagy részét támogatja. Fejlesztési szempontból pedig ugyanaz a T-SQL, mint az on-premises SQL Server esetén, néhány apróbb különbség van a két változat között.

[24] A vállalatnál, helyben telepített SQL Serverek

Az SQL Azure tulajdonképpen egy módosított SQL Server motor, amely kiemelkedően magas rendelkezésre állást és hibatűrést biztosít a felhasználóknak.

Nézzük meg, milyen előnyei vannak annak, hogy az adatbázisunk a felhőben van:

- ✔ Nincs szükség hardware-re.
- ✔ Nincs fizikai telepítés.
- ✔ Licenszeket nem kell beszereznünk és/vagy megvásárolnunk.
- ✔ Patchek és frissítések telepítése automatikusan történik.
- ✔ Magas rendelkezésre állású és hibatűrő a rendszer.
- ✔ Az adatbázisok mérete rugalmas, egyszerűen csökkenthető vagy épp növelhető az üzleti igényeknek megfelelően.
- ✔ A meglévő SQL Server-es eszközökkel kezelhető (pl. SSMS).
- ✔ T-SQL támogatás.
- ✔ Annyit fizetünk érte, amennyit használjuk.
- ✔ Könnyű költségelszámolás.

Fontos megjegyeznünk, hogy az SQL Azure adatbázisokat nemcsak Windows Azure-ban hosztolt alkalmazásból tudjuk elérni. Akár a világ távoli pontjáról is ugyanúgy használni tudjuk az adatbázist. Viszont azzal tisztában kell lennünk, hogy az SQL Azure funkcionalitásban bizonyos téren kevesebbet tud, mint az on-premises SQL Server. Például: Analysis service vagy Online Analytical Processing (OLAP) még nem érhető el az SQL Azure jelenlegi változatában. De korábban is voltak olyan funkciók, amik az első változatba nem kerültek bele, később viszont elérhetővé váltak. Ilyen például az SQL Azure Reporting. Ma már ez a funkció elérhető, és mivel ez online platform, ezért a frissítés is egyszerűen történik.

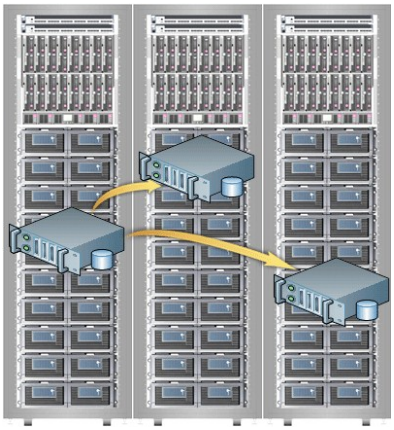
Az alábbi lista az SQL Azure fontosabb hiányosságokat sorolja fel. Abban az esetben, ha számunkra ezek nélkülözhetetlen funkciók, akkor ne válasszuk az SQL Azure-t!

- ■ Elosztott tranzakciók
- ■ Elosztott lekérdezések
- ■ FILESTREAM Data
- ■ Beépített Full-Text Search
- ■ CLR
- ■ Service Broker
- ■ Fizikai szerver és katalógus
- ■ Adatbázis Mirroring
- ■ Extended Stored Procedures
- ■ Tábla particionálás

További hiányosságok ezen az oldalon találhatók.

Az SQL Azure rugalmas, skálázható és biztonságos rendszer. Mit is jelent ez?

Az egyértelmű, hogy rugalmas és skálázható, hiszen az igényeinknek megfelelően változtathatjuk az adatbázisaink méretét, és annak függvényében fizetünk, amennyit felhasználtunk. De mitől biztonságos a rendszer? Az SQL Azure-ban minden adatbázisról kettő darab replika készül. Abban az esetben, ha az egyik kiesik, annak szerepét a következő átveszi. Ekkor ismét készül egy replika, hogy mindenképpen három példány legyen az adott adatbázisból. Így ha hiba történik, az adataink mindig rendelkezésre állnak.



Gyakran merül fel viszont a kérdés, hogy hogyan tudunk biztonsági mentést készíteni az adatbázisunkból. Hisz az on-premise megoldásoknál ez triviális dolog. Az SQL Azure esetében jelenleg még nincs erre beépített lehetőség. Persze saját megoldást készíthetünk, amivel lekérdezzük az adatbázis tartalmát, majd azt eltároljuk egy fájlban, de az mégse ugyanaz, mint egy bak fájl.

12.1 Árazás

Az SQL Azure-nak nagyon kedvező díjszabása van. Az első változat árazása nagyon egyszerű volt, hisz minden gigabájt 9.99$ volt. Tehát egy 5 gigabájtos adatbázis havi szintű fenntartása 49.95$-be került, egy 150 gigabájtos adatbázisé pedig 499.95$ volt. Ez sem volt drága megoldásnak tekinthető, de az árak 2012 februárjában drasztikus csökkenésen mentek keresztül. Bizonyos csomagok esetén 75%-kal is olcsóbb lett az SQL Azure adatbázisok használata, mint korábban.

Az alábbi táblázat a régi és az új adatbázis árakat mutatja be.[25]

Tárterület	Előző ár	Új ár	Új ár/terület	Csökkentve (%)
5	49.95$	25.99$	5.20	48%
10	99.99$	45.99$	4.60	54%
25	299.97$	75.99$	3.04	75%
50	499.95$	125.99$	2.52	75%
100	-	175.99$	1.76	65%
150	-	225.99$	1.51	55%

[25] Fontos megjegyezni, hogy SQL Azure adatbázisok ára változhat, vásárlás előtt mindig tájékozódjunk az aktuális árakról a http://windows.azure.com oldalon!

A 2012. februári frissítésben ezenkívül bevezetésre került egy extra kicsi adatbázis is. Ha szeretnénk, immár 100 megabájtos adatbázis is a rendelkezésünkre állhat, ráadásul nem is akárhogy. Ugyanis az adatbázis létrehozásánál látni fogjuk, hogy a web edition-nél nincs 100 megabájtos opció, de ha a létrehozott adatbázisban 100 megabájtnál kevesebb adatot tárolunk, akkor a havidíj a 9.99$ helyett csak 5$ lesz. Abban az esetben, ha meghaladjuk a 100 megabájtos korlátot, akkor az 1 gigabájtos csomag árai lesznek érvényesek. Ez egy rendkívül jó és költséghatékony megoldás.

Átlagos kis és közepes projekteknél nem szoktak 1 gigabájtnál nagyobb adatbázisokat használni. Gondoljunk bele, hogy szükségünk volna a projektünkhöz egy adatbázisra, ami mindig rendelkezésre áll, hibatűrő, és ha kell, akkor skálázható! Ma már havi 9.99$ -ért (vagy kevesebbért) kaphatunk egy ilyen adatbázist. Ha ezt a saját cégünknek kellene üzemeltetni a mai árak mellett, a szerver áramfogyasztása is drágább lenne. Nem számolva a hardver, licenszek, patchek és egyéb üzemeltetési költségeket.

Most, hogy tudjuk, mi az az SQL Azure, és milyen költségei vannak, nézzük meg, hogy hogyan is kell használni azt!

13 SQL Azure adatbázis szerver létrehozása

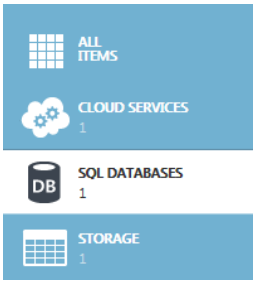
Nézzük meg, hogy hogyan hozzunk létre egy SQL Azure szervert és adatbázist! Mielőtt belekezdenénk, rendelkeznünk kell egy Windows Azure fiókkal. Ha még nem rendelkezünk ilyen felhasználói fiókkal, akkor kérhetünk egy trial fiókot. (További részletek a Trial-ról)

Ha már van Windows Azure fiókunk, akkor kövessük az alábbi lépéseket.

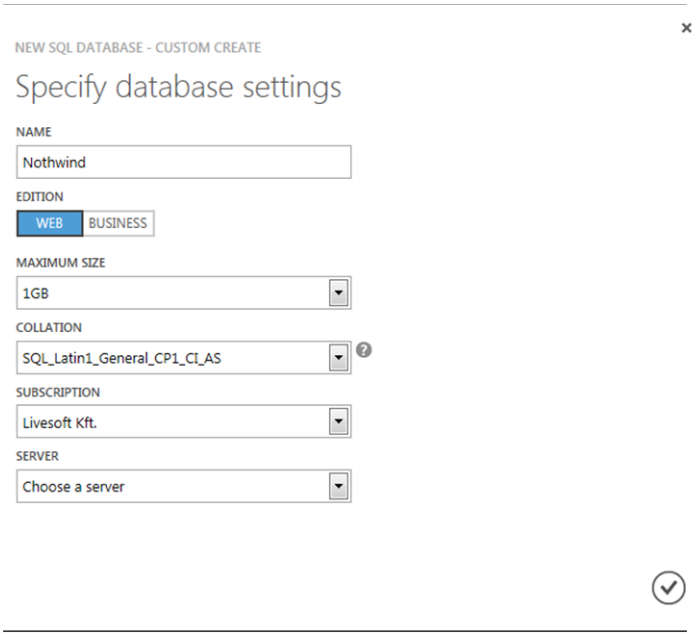
1. Jelentkezzünk be a Windows Azure fiókunkkal (Live ID) a Windows Azure Management Portálra (http://azure.com)!

   Jelenleg kétféle management portál érhető el. A korábbi a Silverlight alapú, az újabb pedig HTML5 alapú portál. A HTML5 alapú portál jelenleg Preview változatban érhető el, de mivel ez fogja leváltani rövidesen a Silverlight alapú portált, ezért most ezt mutatjuk be.

2. A ba loldalon található navigációs sávon kattintsunk az **SQL DATABASES** menüpontra.



3. Következő lépésként kattintsunk az oldal alján található **Add** gombra! Ekkor felugrik a **New SQL Database** ablak.

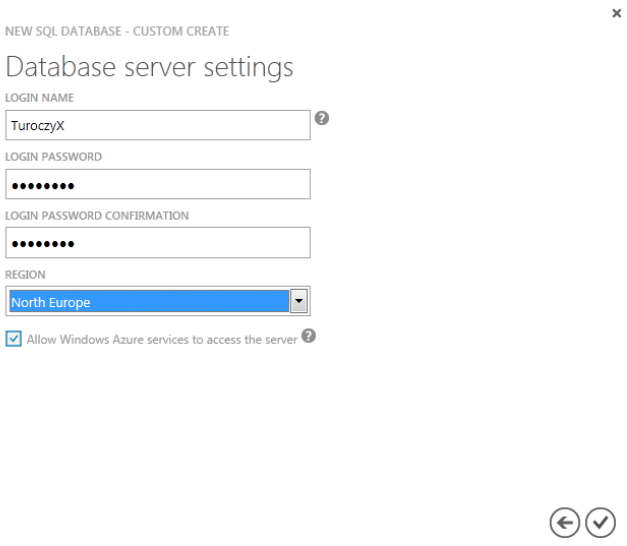


4. A megjelenő ablakban adjuk meg az adatbázisunk nevét (*Name*)! Ezt követően válasszuk ki, hogy mekkora legyen az adatbázisunk mérete! Kétféle változat létezik, amelyek csak a kiválasztható adatbázis méretében különböznek. Létezik egy Web Edition a kis adatbázisok számára, valamint van egy Business Edition a nagyobb adatbázisok számára. A Web edition-ben 1 gigabájtos és 5 gigabájtos adatbázisok közül választhatunk. (*Ne feledjük, ha 100 megabájtnál kevesebbet használunk, akkor csak 5$-t kell fizetnünk!*) A Business edition-ben jelenleg 10 – 150 gigabájtos adatbázisok közül választhatunk. A Collation-nél beállíthatjuk, hogy milyen karakterkódolást használjunk az adatbázisnál. Mivel most magyar nyelvterületen vagyunk, az alapértelmezett jó lesz a számunkra. Ezt követően kiválaszthatjuk, hogy melyik előfizetéshez szeretnénk az adatbázist rendelni. Ha több van, választhatunk közöttük. A Server menüpont alatt kiválaszthatjuk, hogy egy új adatbázis szerveren szeretnénk ezt az adatbázis létrehozni, vagy egy már meglévő adatbázis szerverhez szeretnénk hozzáadni. Mi most új adatbázist szeretnénk létrehozni, így válasszuk ki a **New SQL Database Server**-t a lenyíló listából, majd kattintsunk a tovább gombra! (Balra nyíl)

5. Ebben a pillanatban megjelenik a **Database server settings** ablak. Itt először meg kell adnunk az adminisztrátori felhasználónevünket és jelszavunkat.

Fontos: nem lehet a felhasználónév: sa, admin, administrator, root, guest, dbmanager és loginmanager! Figyeljünk arra is, hogy erős jelszót határozzunk meg!

6. Ezt követően válasszuk ki a számunkra megfelelő régiót. Általában a hozzánk közelebb eső régiót célszerű választani. A régió kiválasztást jól fontoljuk meg, ugyanis ezt megváltoztatni később nem lehet!



A régió kiválasztása a dátumot is befolyásolja!

Például: Ha van egy adatbázisunk a West Europe adatközpontban és meghívjuk a

GETDATE() függvényt, akkor az adatközpont helyi idejét adja meg, ami 2 óra eltérést eredményez számunka, akik Magyarországról használnák az adatbázist. A dátumokat célszerű UTC-ben tárolni.

7. Az utolsó menüpont egy checkbox, ami alapvetően be van kapcsolva. Itt befolyásolhatjuk, hogy az általunk létrehozott adatbázist el lehessen érni egy tetszőleges azure szolgáltatásból, beleértve a saját szolgáltatásunkat is.

8. Ha ezekkel a lépésekkel megvagyunk, kattintsunk a Finish (pipa) gombra, és néhány másodpercen belül az általunk konfigurált adatbázis szerver és adatbázis el fog készülni.

9. Ezt követően a Management Portálon láthatjuk az új SQL Azure szerverünket. Az SQL szerverünk kap egy véletlen nevet, ami ebben az esetben az **e7m4zvqafx**. Ezt az adatbázis szervert majd az e7m4zvqafx.database.windows.net címen érhetjük el.

Egy adatbázis szerveren 149 darab adatbázis hozható létre. Ár szempontjából mindegy, hogy egy új adatbázis szerveren hozunk létre egy új adatbázist, vagy egy szerveren több adatbázist tárolunk. Mindig csak az adatbázisokért fizetünk, nem pedig az adatbázis szerverekért. Fontos megjegyezni továbbá, hogy csak az általunk létrehozott adatbázisért kell fizetnünk, a master adatbázisért, valamint a log fájlokért nem kell külön fizetnünk.

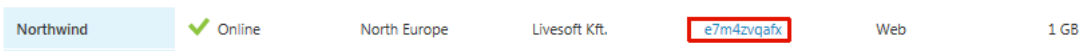
13.1 Tűzfalszabályok

Az elkészített adatbázist jelenleg még nem tudnánk elérni. Ahhoz, hogy az elkészített adatbázist használjuk, meg kell mondanunk, hogy az SQL Azure szerverünket milyen IP cím tartományból érhetjük majd el. Megadhatunk egy IP címet vagy akár IP cím tartományt is. Nézzük meg, hogyan!

1. A baloldalon található navigációs sávon kattintsunk az **SQL DATABASES** menüpontra!

2. Az SQL Databases ablakban megjelennek az adatbázisaink. A **Server** oszlop alatt láthatjuk, hogy az adott adatbázis melyik szerverhez tartozik.

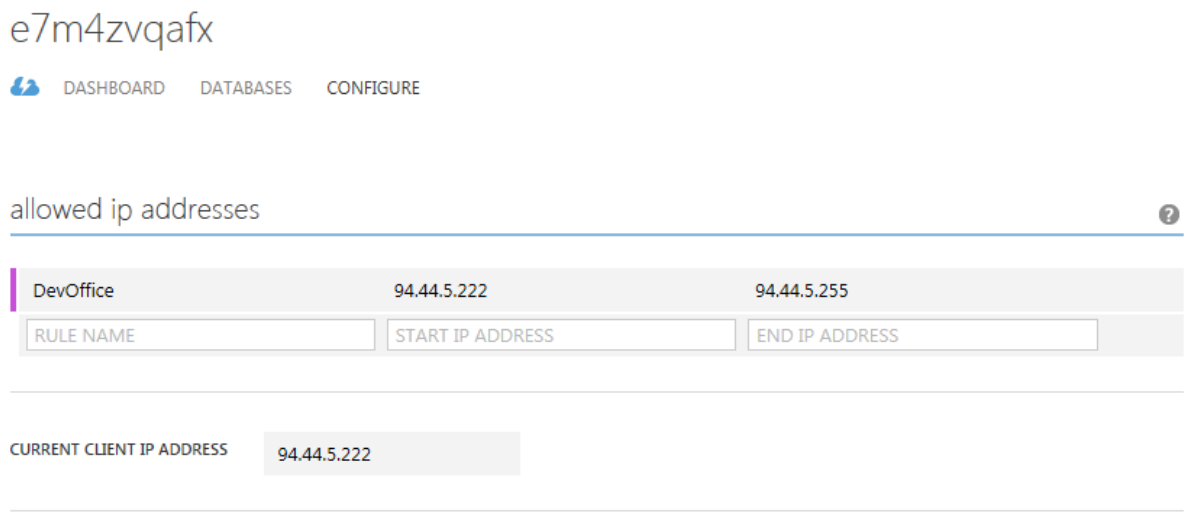
3. Ahhoz, hogy tűzfalszabályt határozzunk meg, kattintsunk az általunk kiválasztott szerver linkjére!



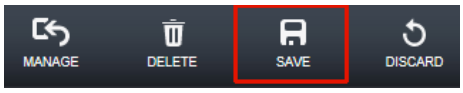
4. A megjelenő oldalon válasszuk ki a **CONFIGURE** menüpontot!

5. Az oldalon meg kell adnunk a tűzfalszabály nevét, ez a név tetszőleges lehet. Jelen esetben a DevOffice nevet adtuk meg a szabályunk számára. Ezenkívül kell egy IP tartományt is megadnunk, amelyből az adatbázist elérhetjük. Ez lehet egy IP cím

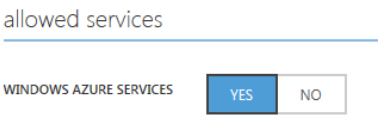
tartomány, de lehet akár csak egy IP cím is. Ebben az esetben a start ip address megegyezik az end ip address értékével. Ha azt akarjuk, hogy a világ bármely pontjáról elérhető legyen az adatbázisunk, akkor a start ip address-nek 1.1.1.1 –es, míg az end ip address-nek a 255.255.255.255 ip címet kell megadnunk. Természetesen több szabályt is létre tudunk hozni.



6. Ha megadtuk a szabályainkat, kattintsunk a **Save** gombra az oldal alján!



Az, hogy megadtunk egy IP cím tartományt, ahonnan elérhetjük az adatbázist, még nem jelenti azt, hogy az Azure alkalmazásokból is el tudjuk érni. Ahhoz, hogy az azure alkalmazás is elérje ezt az adatbázist, az **Allow services** menüpontnak engedélyezve kell lennie. Ezt az adatbázis létrehozásnál is megadhatjuk.



A tűzfalszabályokat a későbbiek folyamán bármikor módosíthatjuk vagy törölhetjük.

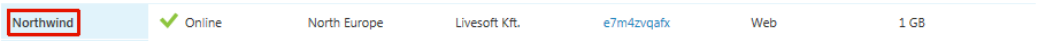
Nemcsak a Management Portálról tudjuk a tűzfalszabályainkat módosítani, lehetőségünk van akár PowerShellből is új szabályokat meghatározni vagy a meglévőket szerkeszteni, törölni. Ehhez le kell töltenünk a Windows Azure PowerShell Cmdlets szkriptet a http://wappowershell.codeplex.com oldalról.

13.2 Adatbázis skálázása

Használat közben bármikor előfordulhat, hogy a meglévő adatbázisunk mérete már nem elégséges és szeretnénk módosítani annak méretét. Természetesen erre is van lehetőségünk.

1. A bal oldalon található navigációs sávon kattintsunk az **SQL DATABASES** menüpontra!

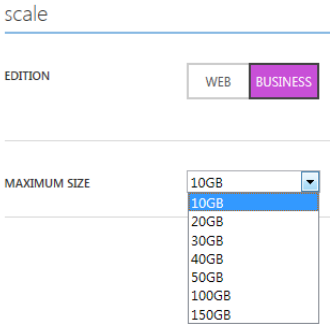
2. Ahhoz, hogy az adatbázisunk méretét változtassuk, kattintsunk az általunk kiválasztott adatbázis nevére!



3. A megjelenő oldalon a kiválasztott adatbázis Dashboardját láthatjuk, ahol a felhasznált tárterületet, valamint az ehhez kapcsolódó részletes adatokat vehetjük szemügyre.

4. Ezen az oldalon kattintsunk a **Scale** menüpontra!

5. Itt állítsuk be, hogy mi legyen az új mérete az adatbázisnak!



6. Ha kiválasztottuk a megfelelő méretet, kattintsunk a **Save** gombra! A folyamat néhány másodpercet vesz igénybe. Ha most visszatérünk a Dashboard-ra, láthatjuk az adatbázis kihasználtságánál, hogy nem 1 gigabájtnyi, hanem 10 gigabájtnyi tárterület áll a rendelkezésünkre.

Az adatbázisok mérete nemcsak felfelé, hanem akár lefelé is módosítható!

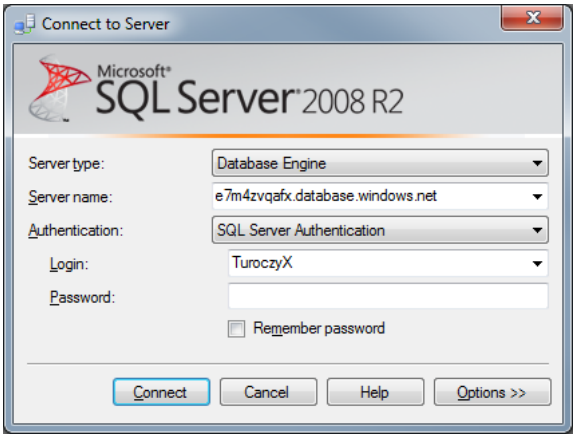
14 Adatbázisok elérése

Kész az adatbázisunk, így hát itt az ideje elérni és használatba venni. Ehhez a már megszokott eszközöket is alkalmazhatjuk, például elérhetjük az SQL Azure adatbázis szerverünket az SQL Server Management Studio 2008 R2 –vel is.

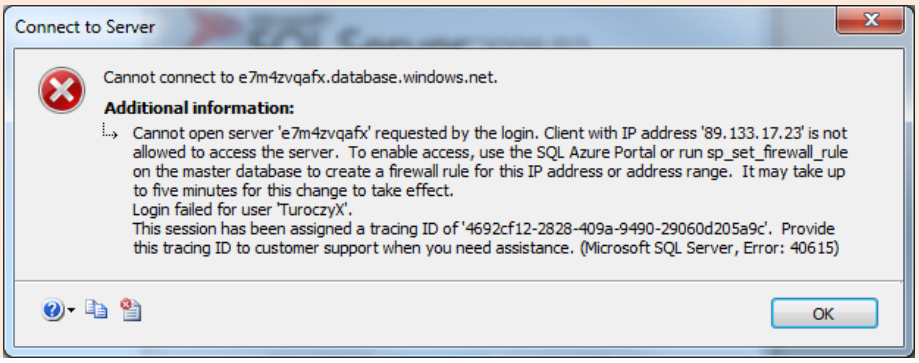
Indítsuk el az SQL Server Management Studio 2008 R2-t alkalmazást! (Amennyiben nincs telepítve, legegyszerűbben a Web Platform Installer-rel telepíthetjük http://www.microsoft.com/web/downloads/platform.aspx).

1. Az alkalmazás indulásakor felugrik a **Connect to Server** ablak. Ebben az ablakban a Server name-hez meg kell adnunk az SQL Azure szerverünk címét. Ez a cím mindig a szerver neve <*servername*>.database.windows.net címen érhető el. A <*servername*> a szerverünk egyedi nevét jelöli, ami ebben a példában az e7m4zvqafx.

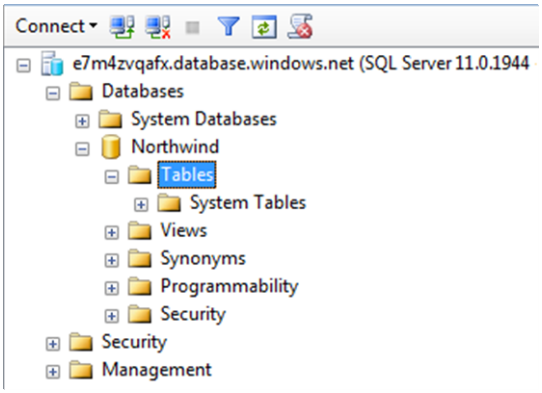
2. Az Authentication-t állítsuk át **SQL Server Authentication**-re, majd adjuk meg a felhasználónevünket és jelszavunkat. A Windows Authentikáció nem támogatott!



FONTOS! Mindig figyeljünk oda a tűzfalszabályokra! Abban az esetben, ha helytelenül vannak beállítva, az SQL Azure szerverünket nem tudjuk elérni! Ebben az esetben a 40615 hibaüzenetet fogunk kapni.



3. Ha készen vagyunk, kattintsunk a **Connect** gombra! Ha sikerült a kapcsolódás, akkor az Object Explorerben a meglévő adatbázisunk fogad.



Innentől kezdve az Object Explorerből úgy kezeljük az adatainkat, mint amit az SQL Server-nél megszoktunk. Arra viszont figyeljünk, hogy ilyenkor az SQL Server Management Studiónk kisebb funkcionalitással bír! Például, ha táblát szeretnénk létrehozni, azt jelenleg csak szkriptből tehetjük meg. Nincsenek csillogó varázslóink, mint az on-primese SQL Server-nél.

14.1 Adatbázis létrehozás és menedzselés

Ha sikerült kapcsolódunk a master adatbázishoz, lehetőségünk nyílik új adatbázist létrehozni, vagy a már meglévőket módosítani vagy épp törölni. Az alábbiakban megnézzük, hogy a leggyakrabban használt műveleteket hogy érjük el Management Studio-ból.

Figyeljünk arra, hogy valóban a master adatbázishoz csatlakoztunk!

Az **Object Explorer**-ben kattintsunk a **Databases**-en belül a **System Databases**-re, majd jobb egérgombbal a **master**-re, és a helyi menüből válasszuk ki a **New Query** parancsot! A megjelenő **Query Window**-ba írhatjuk a szkriptjeinket.

Adatbázist Management Studio-ból is létrehozhatunk, nem fontos ehhez a Management Portálra látogatnunk. Csakúgy, mint a helyi SQL Serverek esetén, itt is a CREATE DATABASE parancs lesz a segítségünkre. Itt meg kell adnunk az adatbázisunk méretét és változatát (edition) is. Például ha egy 1 gigabájtos Web Edition-ös adatbázist szeretnénk létrehozni, akkor az alábbi szkriptet kell leírnunk:

```
CREATE DATABASE DevTest (MAXSIZE=1GB, EDITION='web');
```

Abban az esetben, ha nem megfelelő mérettel szeretnénk lefuttatni a szkriptet, hibát kapnánk. Például: Ha 1 gigabájt helyett 2 gigabájtot írnánk, ami a Web Edition-ben nem található, az alábbi hibaüzenetet kapnánk:

```
The edition 'web' does not support the database max size '2 GB'.
```

Ha pedig az edition-t rontjuk el, akkor az alábbi hibaüzenetet fogjuk kapni:

```
Invalid value given for parameter EDITION. Specify a valid parameter value.
```

Viszont ha nem adunk meg paramétereket, akkor alapértelmezetten egy Web Edition-ös 1 gigabájtos adatbázist hozunk létre.

```
CREATE DATABASE DevTest
```

Ha módosítani szeretnénk az adatbázisunk méretét és/vagy típusát, akkor azt az ALTER DATABASE paranccsal tehetjük meg. Az alábbi példában a már korábban létrehozott DevTest adatbázis méretét módosítjuk 1 gigabájtról 5 gigabájtra.

```
ALTER DATABASE DevTest

MODIFY (MAXSIZE=5GB, EDITION='web')
```

Abban az esetben, ha tovább már nincs szükségünk egy adatbázisra, akkor a DROP DATABASE paranccsal törölhetjük az adatbázisunkat. Az alábbi példában a DevTest adatbázist töröljük (*dbmanagerek törölhetik az adatbázist*).

```
DROP DATABASE DevTest;
```

Ha meg szeretnénk nézni a rendelkezésre álló adatbázisainkat, akkor a master **sys.databases** nézetét kell lekérdeznünk.

```
SELECT * FROM sys.databases;
```

	name	database_id	source_database_id	owner_sid	create...	compatibility_level	collation_name	user_access	user_access_desc
1	master	1	NULL	0x010600...	2012-...	100	SQL_Latin1_General_CP1_CI_AS	0	MULTI_USER
2	Northwind	4	NULL	0x010600...	2012-...	100	SQL_Latin1_General_CP1_CI_AS	0	MULTI_USER
3	DevTest	5	NULL	0x010600...	2012-...	100	SQL_Latin1_General_CP1_CI_AS	0	MULTI_USER

Mint láthatjuk, az adatbázis menedzsmentben nem sok újdonság van az on-primeses megoldásokhoz képest. Néhány aprósággal érdemes tisztában lenni, de aki már otthonosan mozog az SQL Server világában, annak az SQL Azure használata nem jelent új kihívásokat.

14.2 Hozzáférések kezelése

Hozzáférési jogosultságokat is megadhatunk az SQL Azure adatbázisunk számára. A már korábban megismert CREATE LOGIN, ALTER LOGIN és a DROP LOGIN parancsok használatával menedzselhetjük az adatbázis szerverünk hozzáférését. Szintén fontos megjegyezni, hogy ehhez a **master** adatbázishoz kell csatlakoznunk.

A CREATE LOGIN paranccsal hozhatunk létre egy új szerver szintű hozzáférést. Az alábbiakban egy Attila nevű felhasználót hozunk létre, akinek a jelszava a Password1 lesz.

```
CREATE LOGIN Attila WITH password='Password1';
```

Ezzel egy szerver szintű felhasználót hoztunk létre. Ha viszont adatbázis szintű felhasználót szeretnénk létrehozni, akkor a CREATE USER parancs lesz a segítségünkre. Minden logint a master adatbázisba kell létrehoznunk, de ahhoz, hogy csatlakozni tudjunk egy meghatározott

adatbázishoz, ezt adatbázis szinten kell megtennünk. Figyeljünk arra, hogy a DevTest adatbázison futtassuk le az alábbi parancsot!

```
CREATE USER DevUser FROM LOGIN Attila
```

Ezt követően a DevTest adatbázishoz (és csakis ahhoz) csatlakozhat az Attila nevű felhasználó.

Meghatározhatjuk, hogy az adott felhasználó fiók milyen elemekhez férhet hozzá. Ehhez az sp_addrolemember tárolt eljárás lesz a segítségünkre. Ebben az esetben a DevUser-t hozzáadjuk a **db_datareader** szerepkörhöz.

```
exec sp_addrolemember 'db_datareader', 'DevUser';
```

Az ALTER LOGIN paranccsal módosíthatunk egy már meglévő logint, melyet a master adatbázison kell lefuttatnunk. Az alábbiakban az Attila nevű felhasználó jelszavát cseréljük le.

```
ALTER LOGIN Attila
WITH PASSWORD = 'newPassword1'
OLD_PASSWORD = 'Password1';
```

Ha egy logint törölni szeretnénk, akkor a DROP LOGIN paranccsal tehetjük meg. Ezt a parancsot is a master-en kell futtatnunk. Ha szerver szinten törlünk, akkor az asszociált adatbázis szintű felhasználói fiókok is törlődnek.

```
DROP LOGIN Attila
```

Ha meg akarjuk nézni, hogy milyen loginjaink vannak, a **sys.sql_logins** segítségével lekérdezhetjük a rendszerben lévő összes logint. Ezt a parancsot is a master-en kell futtatnunk.

```
SELECT * FROM sys.sql_logins
```

Az eredmény:

	name	principal_id	sid	type	type_desc	is_disabled
1	TuroczyX	1	0x01060000...	S	SQL_Login	0
2	Attila	2	0x01060000...	S	SQL_Login	0

14.3 Tábla létrehozása, lekérdezése

Láthattuk, hogyan készítsünk adatbázist, hogyan hozzunk létre felhasználó fiókot, de még nem hoztunk létre egy táblát sem. Jelöljük ki a DevTest adatbázist az **Object Explorer**-ben, majd kattintsunk jobb egérgombbal, és válasszuk ki a **New Query** menüpontot! (*Nem használhatjuk a USE parancsot!*) A megjelenő szerkesztő felületre írjuk a következőt:

```
CREATE TABLE Developers(ID INT, Name VARCHAR(20))
```

```
INSERT INTO Developers VALUES(1, 'Attila')
```

Ezzel a szkripttel egy baj van: mégpedig az, hogy nem fog működni. Látszólag minden rendben van, de mégsem. A táblát ugyan létrehozzuk, de a szkript az insert-nél „elhasal” az alábbi hibával:

```
Msg 40054, Level 16, State 1, Line 2
Tables without a clustered index are not supported in this version of SQL Server. Please
create a clustered index and try again.
```

Ez azért van, mert mindenképp szükségünk van egy clustered index-re. Érdemes kihasználnunk, hogy az SQL Azure (akárcsak az Sql Server) alapértelmezetten létrehoz egy clustered indexet az elsődleges kulcsra. Módosítsuk a szkriptünket az alábbiak szerint:

```
CREATE TABLE Developers(ID INT PRIMARY KEY, Name VARCHAR(20))

INSERT INTO Developers VALUES(1, 'Attila')
```

Ebben az esetben a táblánk szintén elkészül, és az insert is le fog futni.

A Select, Insert, Update műveletekben nagy változás nincs. A lekérdezések és az adatmódosítások a már megszokott módon történnek.

```
SELECT * FROM Developers;
```

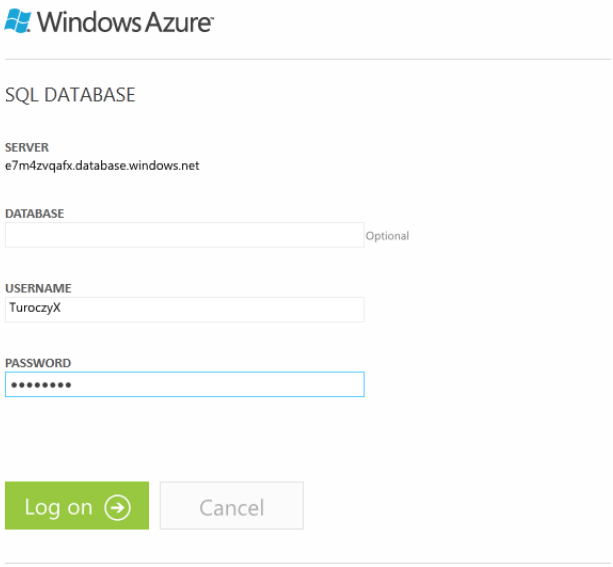
	CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Region
37	HUNGC	Hungry Coyote Import Store	Yoshi Latimer	Sales Representative	City Center Plaza 516 Main St.	Elgin	OR
38	HUNGO	Hungry Owl All-Night Grocers	Patricia McKenna	Sales Associate	8 Johnstown Road	Cork	Co. ...
39	ISLAT	Island Trading	Helen Bennett	Marketing Manager	Garden House Crowther Way	Cowes	Isle ...
40	KOENE	Königlich Essen	Philip Cramer	Sales Associate	Maubelstr. 90	Brandenburg	NULL
41	LACOR	La corne d'abondance	Daniel Tonini	Sales Representative	67, avenue de l'Europe	Versailles	NULL
42	LAMAI	La maison d'Asie	Annette Roulet	Sales Manager	1 rue Alsace-Lorraine	Toulouse	NULL
43	LAUGB	Laughing Bacchus Wine Cellars	Yoshi Tannamuri	Marketing Assistant	1900 Oak St.	Vancouver	BC
44	LAZYK	Lazy K Kountry Store	John Steel	Marketing Manager	12 Orchestra Terrace	Walla Walla	WA
45	LEHMS	Lehmanns Marktstand	Renate Messner	Sales Representative	Magazinweg 7	Frankfurt a....	NULL
46	LETSS	Let's Stop N Shop	Jaime Yorres	Owner	87 Polk St. Suite 5	San Franci...	CA
47	LILAS	LILA-Supermercado	Carlos González	Accounting Manager	Carrera 52 con Ave. Bolívar ...	Barquisimeto	Lara

15 SQL Azure Management Portál

Az SQL Azure adatbázisunkat nemcsak SQL Server Management Studio segítségével kezelhetjük, hanem akár egy webes portál segítségével is.

1. A Management Portálon navigáljunk el az előfizetésünkhöz tartozó adatbázishoz, majd kattintsunk a Manage gombra! (A webes menedzsment felületet közvetlen módon is elérhetjük. Ennek a címe a https://*.database.windows.net, ahol a * az adatbázisunk nevét jelöli. Tehát ebben az esetben a teljes cím a https://e7m4zvqafx.database.windows.net).

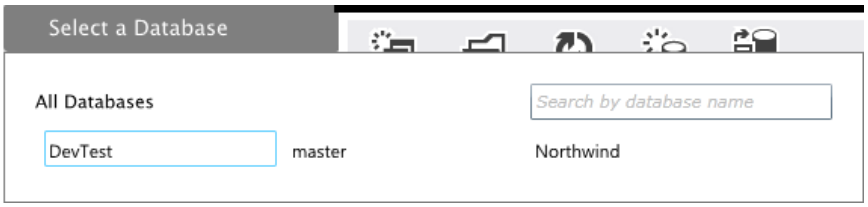
2. Amikor elnavigálunk az adott címre, egy Silverlightos alkalmazás fog elindulni. Itt meg kell adnunk a felhasználó nevünket, valamint jelszavunkat. (Opcionálisan az adatbázis nevét). Majd kattintsunk **a Log On** gombra!



   Ne feledjük engedélyezni azt, hogy az adatbázist azure szolgáltatások is elérhessék!

3. Egy Metro UI-os webes adatbázis menedzser fogad. Ez az SQL Server Management Studio kistestvére, amit bármikor, bárhonnan elérhetünk (Tűzfalszabályoknak megfelelően). Fontosabb funkciókat erről a felületről is el tudunk érni.

4. Nézzük meg, mire jó ez a felület! Ha a kapcsolódásnál nem határoztunk meg adatbázist, és rendszer szintű felhasználók vagyunk, akkor a bal felső sarokban kiválaszthatjuk, hogy melyik adatbázissal szeretnénk dolgozni. Most válasszunk ki egy adatbázist, legyen a **Northwind**!

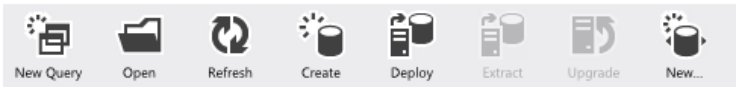


Látható az is, hogy egy keresés mező is rendelkezésünkre áll, hogy gyorsabban megtaláljuk a szükséges adatbázisunkat. Ez főleg nagy adatbázis számnál jöhet jól.

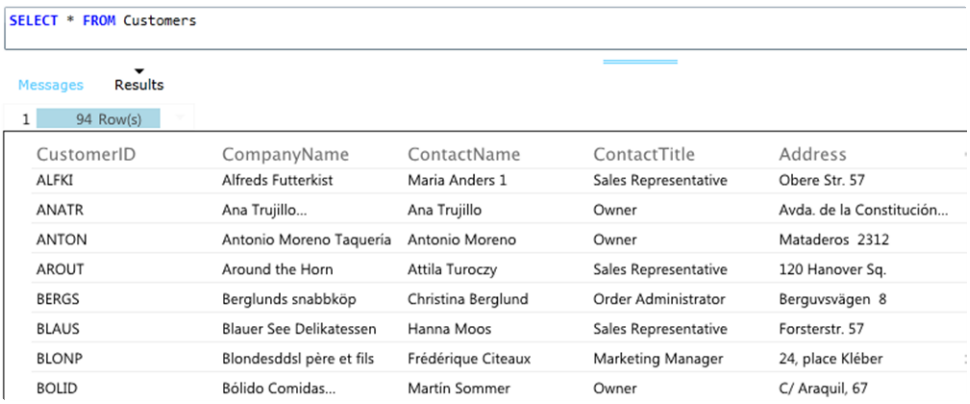
5. Amint kiválasztottuk az adatbázist, egy összegző nézet jelenik meg.

Itt láthatunk minden fontosabb információt az adatbázisunkról. Mikor készítettük, mi a Collation beállítása, hány aktív kapcsolat van vagy mekkora az adatbázis maximális és felhasznált mérete.

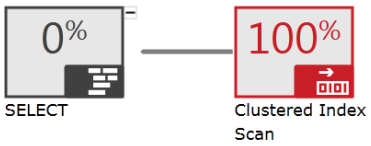
6. A felső menüsorban a legfontosabb funkciókat találjuk. Új lekérdezéseket írhatunk, betölthetünk sql szkripteket, vagy akár új adatbázist is készíthetünk.



7. A **New Query** gombra kattintva betöltődik egy szkript szerkesztő. Ide írhatjuk saját lekérdezéseinket, természetesen ez a felület is rendelkezik kódkiemeléssel.



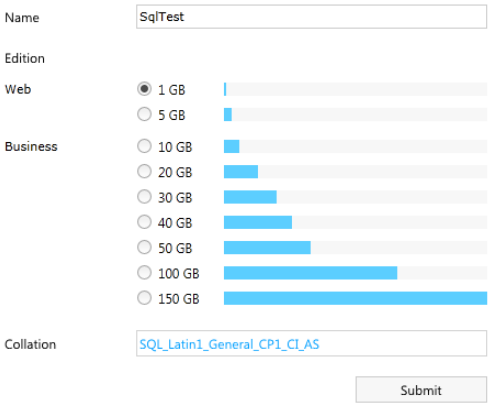
8. Lekérdezéseink Query Plan-jét is megtekinthetjük (**Actual plan**).



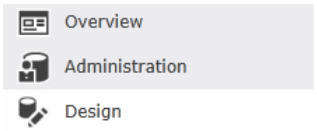
Nemcsak lekérdezéseket fogalmazhatunk meg az SQL Azure Management felületén, hanem akár új adatbázist is készíthetünk, és táblákat is menedzselhetünk, sőt akár Data-Tier alkalmazásokat is feltölthetünk.

1. Kattintsunk a bal alsó sarokban az **Administration** menüpontra, majd a megjelenő oldalon a **Create** gombra!

2. A megjelenő ablakban meg kell adnunk az adatbázisunk nevét, méretét és collation beállítását. Ha ezt megtettük, kattintsunk a **Submit** gombra!

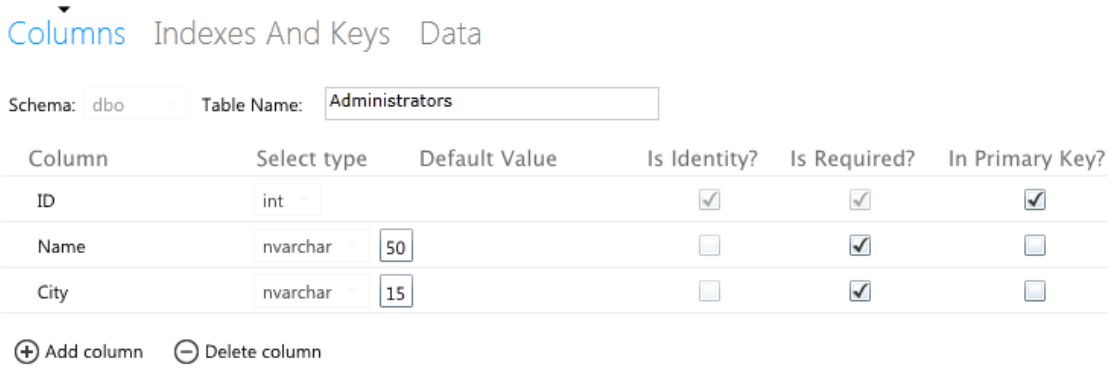


3. Néhány másodperc alatt elkészül az adatbázisunk. Itt nemcsak elkészíthetjük, de akár a táblákat is megszerkeszthetjük. Kattintsunk a bal alsó sarokban a **Design** menüpontra!



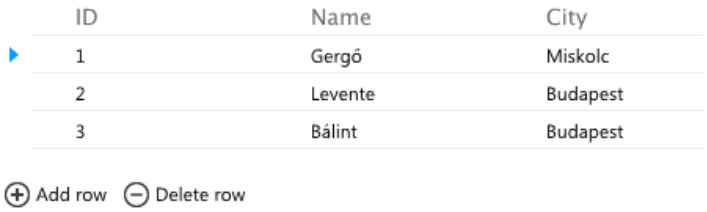
4. Itt táblát, nézetet, valamint tárolt eljárást is létrehozhatunk. Mi most egy egyszerű táblát fogunk létrehozni, kattintsunk a **New table** menüpontra!

5. A megjelenő ablakban adjuk meg a tábla nevét, ami most *Administrators* lesz! Három mező lesz benne: egy ID, ami elsődleges kulcs, az Is Identity kapcsolója true, valamint egy Name mező, illetve egy City mező amelyek nvarchar típusúak. (Méretük lényegtelen.)



6. Ha megvagyunk, kattintsunk a **Save** gombra!

7. A táblánkat létrehoztuk, de ez a tábla még üres. Ha módosítani vagy új adatot szeretnénk hozzáadni, kattintsunk a **Data** menüpontra! Az **Add row** gomb megnyomásával új rekordokat adhatunk a táblánkhoz. Adjunk néhány rekordot a táblánkhoz. A mentés csak akkor következik be, ha a **Save** gombra kattintottunk. Nem ment automatikusan a rendszer.



Láthatjuk, hogy a fontosabb műveleteket elérhetjük webről is. Nem kell hozzá kliensalkalmazást telepíteni. Használata könnyű és gyors. Ez egy jó és szép eszköz, de összetettebb feladatokra az SQL Server Management Studio ajánlott.

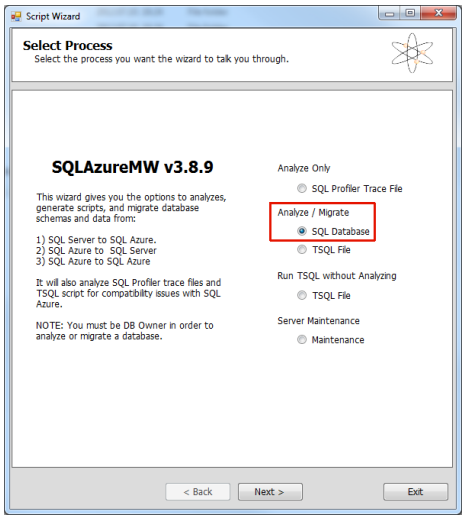
16 SQL Azure Migration Wizard

A SQL Azure Migration Wizard (SQLAzureMW) segítségével egyszerűen tudunk migrálni SQL Server 2005/2008/2012-ből SQL Azure-ba. Az SQLAzureMW analizálja az adatbázisunkat, és ha valamilyen kompatibilitási problémát talál, akkor szól, illetve ha tudja, akkor javítja. Így az SQL Serverünkből az adatbázist könnyedén átemelhetjük az SQL Azure környezetbe. A migrációra a következő lehetőségeink vannak: migrálhatunk SQL Server –ből SQL Azure-ba, SQL Azure-ból helyi SQL Server-re, illetve SQL Azure és SQL Azure között is történhet a migrálás.

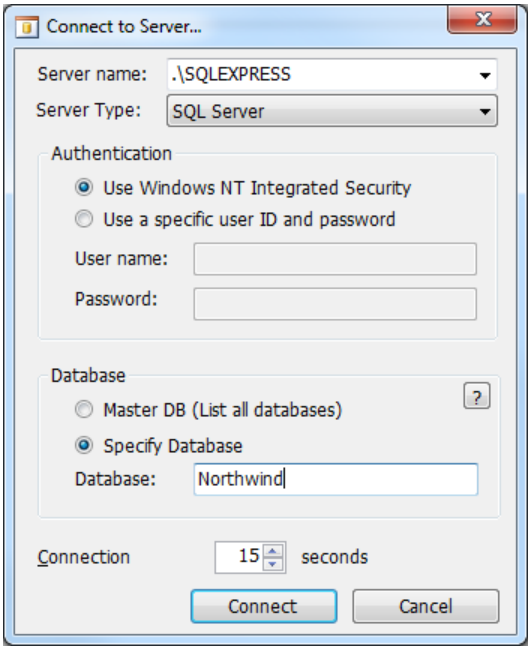
Az SQL Azure Migration Wizardot a http://sqlazuremw.codeplex.com –ról tölthetjük le. Az oldalon nemcsak a termék, hanem annak a forráskódja is letölthető.

Az SQL Azure Migration Wizard jelenleg két változatban érhető el. A v3.X-es változathoz SQL Server 2008 R2 SP1, míg a v4.X-es változathoz SQL Server 2012 kell.

1. Indítsuk el az alkalmazást (SQLAzureMW.exe)!
2. A megjelenő ablakban most válasszuk ki az **Analyze / Migrate** menüpontból az **SQL Database**-t, majd kattintsunk a **Next** gombra!



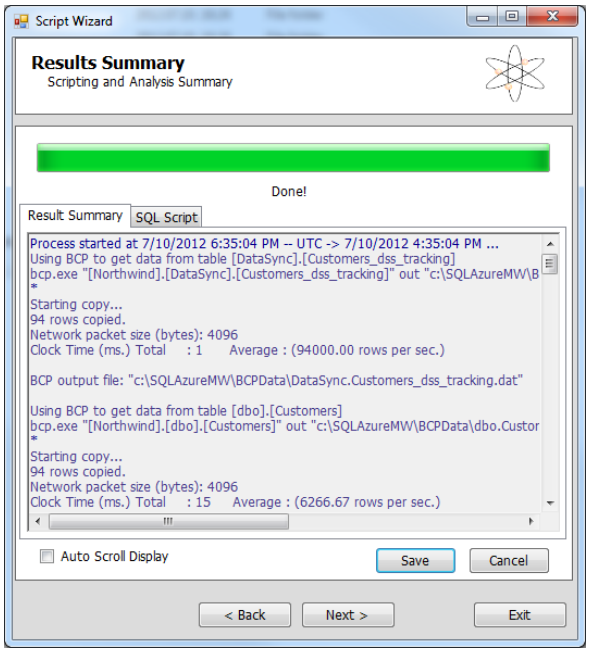
3. A megjelenő **Connect to Server** ablakban adjuk meg a helyi/hálózati SQL Serverünk elérhethőségét! Ez ebben az esetben a .\SQLEXPRESS, valamint határozzuk meg, hogy melyik adatbázist szeretnénk migrálni!



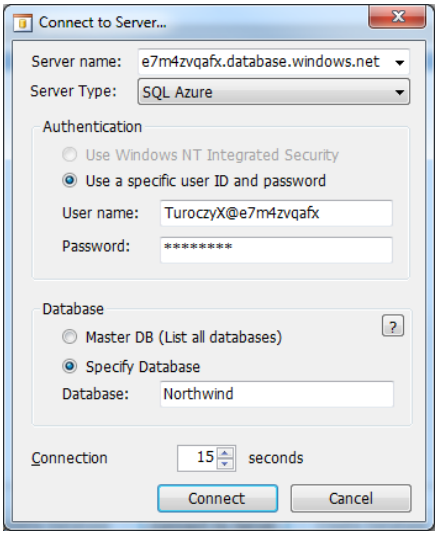
Ha ezzel megvagyunk, kattintsunk a **Next** gombra!

4. A megjelenő ablakban kiválaszthatjuk, hogy mely objektumokat szeretnénk migrálni. Ezek a beállítások most jók számunkra. További beállítási lehetőségeink vannak az Advanced menüpont alatt. Itt állíthatjuk be például azt is, hogy csak a sémát, csak az adatot, vagy a tábla sémát és az adatot együtt szeretnénk migrálni. Kattintsunk a **Next** gombra!
5. Egy **Scirpt Wizard Summary** fogad minket. Összegzi, hogy mit és hogyan szeretnénk migrálni. Ha ez megfelelő számunkra, kattintsunk a **Next** gombra!

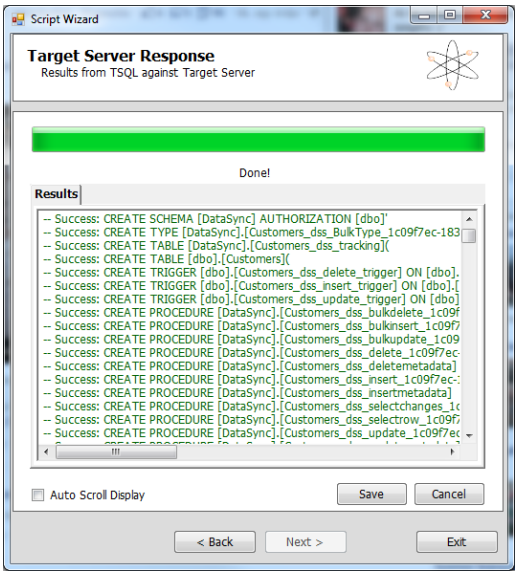
6. Ezt követően elindul az átalakítás. Ez az adatbázis méretétől függően néhány másodperctől kezdve akár több percig is eltarthat. Ha sikeres volt az átalakítás, kattintsunk a **Next** gombra! (Az eredményt kimenthetjük *.rtf-be illetve *.sql-be is.)



7. Majd felugrik a **Connect to Server** ablak. Itt viszont most az SQL Azure-hoz kell kapcsolódnunk. Adjuk meg az SQL Azure kapcsolódási beállításait, a szerver nevét, hozzáférési adatait, az adatbázis nevét, majd kattintsunk a **Connect** gombra!



8. Ha mindennel megvagyunk, kattintsunk a **Next** gombra. Megjelenik egy figyelmeztető ablak, ahol a **Yes** –re kattintva elindul a szkript az SQL Azure-os adatbázisunkon. A folyamat az adatbázis méretétől függően néhány percig is eltarthat. Ha sikerült, és minden rendben zajlott, akkor az alábbi képernyő fogad:



9. Ha most megnézzük az adatbázisunkat, láthatjuk, hogy minden lehetséges adatot átmigráltunk az SQL Azure adatbázisba.

Észrevehetjük, hogy meglévő adatbázisainkat néhány egyszerű lépés segítségével átmozgathatjuk az SQL Azure-os adatbázisba. De ne feledjük, nemcsak SQL Serverből lehet migrálni, lehetőségünk van SQL Azure-ból SQL Server-re, sőt akár SQL Azure és SQL Azure között is.

Ezeket a lépéseket SQL Server Management Studióval is megtehetjük, csak nem segít annyit nekünk, mint az SQL Azure Migration Wizard.
SSMS esetében szkriptet kell generálnunk az adatbázisunkból (*meg kell adnunk, hogy SQL Azure kompatibilis szktipet készítsen*), majd az átalakított szkriptet az SQL Azure adatbázis szerverén le kell futtatnunk.

Természetesen nemcsak SQL Server és SQL Azure között migrálhatunk. Léteznek különböző eszközök, amelyek elősegítik a migrálását akár nem Microsoft-os adatbázisokból is. Ilyen például az SSMA for MySQL, amely a MySQL adatbázis migrálását teszi lehetővé SQL Azure-ba, vagy az SSMA for Access, amely az Access adatbázisokat migrálja az SQL Azure adatbázisunkba. Ezek Microsoft által támogatott eszközök!

17 SQL Azure adatbázis elérése kliens alkalmazásból

Ahhoz, hogy az SQL Azure adatbázisban tárolt adatainkat elérjük egy kliens alkalmazásból, az alkalmazás üzleti logikáján semmit sem kell módosítanunk. Tulajdonképpen csak a „csatlakozási karakterláncot" (Connection String) kell változtatnunk. Egy helyesen megírt .NET alkalmazásnál ráadásul a connection string általában egy app.config (web.config) xml fájlban van letárolva. Így az alkalmazásunkat nem kell újrafordítanunk ahhoz, hogy az adatbázis most már az SQL Azure-ba fusson helyi SQL szerver példány helyett. Az adatok feldolgozásához ugyanazokat az ADO.NET-es objektumokat használhatjuk, mint eddig (SqlCommand, SqlDataReader, SqlConnection stb). Sőt, ha Linq To SQL-t vagy Entity Frameworkot használunk, ott sincs szükség extra munkára, szintén csak a connection string-et kell átírnunk.

Nézzünk erre egy egyszerű példát! Most egy egyszerű konzol alkalmazással érünk el egy adatbázist.

```
using System;
using System.Data.SqlClient;
namespace SqlAzureDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                using (SqlConnection conn = new SqlConnection())
                {
                    conn.ConnectionString = "ConnectionString";
                    using (SqlCommand cmd = new SqlCommand())
                    {
                      cmd.Connection = conn;
                      cmd.CommandText = "SELECT CustomerID FROM Customers";
                      conn.Open();
                      using (SqlDataReader dr =
cmd.ExecuteReader(System.Data.CommandBehavior.CloseConnection))
                      {
                          while (dr.Read())
                          {
                              //Csináljon valamit.
                              Console.WriteLine(dr[0]);
                          }
                          dr.Close();
                      }
                    }
                }
            }
            catch (SqlException ex)
            {
                Console.WriteLine(ex.Message);
                //Valami baj történt
```

```
            }
        }
    }
}
```

A példában az egyszerűség és az átláthatóság kedvéért a Connecton string-et beégetjük a forráskódba. Ezt egy éles alkalmazásnál célszerű egy külső konfigurációs fájlban elhelyezni (App.config, web.config stb.).

Ha a helyi SQL Server Express példányát szeretnénk elérni, akkor a Connection String az alábbiak szerint nézne ki:

```
@"Server=.\SQLEXPRESS;Database=Northwind;Trusted_Connection=Yes";
```

Látható, hogy ez klasszikus connection string, ahol megadjuk az adatbázis szerver elérhetőségét, az adatbázis nevét, valamint a hitelesítő adatokat, ami ebben az esetben megegyezik a Windows rendszerbe bejelentkezett felhasználóéval.

SQL Azure esetén nem lehet Trusted_Connection-t használni, csakis kizárólag felhasználónév jelszó párossal tudjuk hitelesíteni magunkat. Ha a fentebbi connection string-et lecseréljük az alábbira, akkor már nem a helyi sql szerverből fogja lekérdezni az adatokat, hanem az SQL Azure adatbázisunkból. Ebben az esetben a connection string az alábbiak szerint módosul:

```
"Server=e7m4zvqafx.database.windows.net;Database=Northwind;User
ID=TuroczyX;Password=myPassword1";
```

Figyeljük meg, hogy ebben az esetben megadtuk az SQL Azure szerverünk teljes elérési útvonalát,, ami jelenleg az *e7m4zvqafx.database.windows.net*! A Database ugyanaz, mint a helyi példány esetén. Itt határozzuk meg, hogy az adott adatbázis szerveren melyik adatbázist szeretnénk elérni, jelen esetben a Northwind-et. Az authentikáció ebben az esetben csak felhasználónév és jelszó birtokában történhet. Windows authentikációra nincs lehetőség. Ez ugyanúgy történik, mintha a helyi SQL serverbe authentikálnánk. Tehát amit látnunk kell, hogy csak az adatbázis címét, valamint a hitelesítési adatainkat kell módosítani.

Korábban a connection string valamivel összetettebb volt. Például a Server neve előtt egy tcp: prefixum volt, valamint a felhasználónevet is csak úgy adhattuk meg, hogy kiegészítjük a szerver nevével az alábbi módon: TuroczyX@e7m4zvqafx. Mára ez egyszerűsödött, de nézzük meg ugyanezt a példát a régebbi connection string megadásával!

```
"Server=tcp:e7m4zvqafx.database.windows.net;Database=Northwind;User
ID=TuroczyX@e7m4zvqafx;Password=myPassword1";
```

Mind a kétféle megadási módszert alkalmazhatjuk, az SQL Azure elfogadja.

Ha Entity Frameworkot használunk, akkor a connection string az alábbiak szerint módosul:

```
metadata=res://*/Model1.csdl|res://*/Model1.ssdl|res://*/Model1.msl;provide
r=System.Data.SqlClient;provider connection string="data
source=e7m4zvqafx.database.windows.net;initial catalog=Northwind;user
id=TuroczyX;password=myPassword1;"
```

Mint láthatjuk, csak a szerver címe különleges, minden más megegyezik a korábbi connection stringgel.

Figyeljünk arra, hogy ebben az esetben az adatbázisunkhoz tartozó connection string egy XML fájlban lesz letárolva! Mivel csak SQL Authentikációt támogat az SQL Azure, ezért ebben a fájlban szabadon olvasható módon benne lesz a felhasználónevünk-jelszavunk. Ezt természetesen titkosíthatjuk. Az alábbi oldalon további információkat találhat erről: http://msdn.microsoft.com/en-us/library/89211k9b(v=vs.80).aspx.

18 SQL Azure Data Sync

Az SQL Azure Data Sync segítségével a saját on-premise SQL Szerverünk és az SQL Azure között szinkronizálhatjuk az adatainkat. Sőt akár több SQL Azure példány között is történhet szinkronizáció. Ez a technológia a Microsoft Sync Frameworkön alapul. A szinkronizáció kétirányú is lehet. Nézzük meg, hogy hogyan lehet egy szinkronizációs mechanizmust létrehozni! Akár SQL Server Express változatával is szinkronizálhatunk.

Ezt a funkciót a „régi" Silverlightos portálon mutatom be, ugyanis ez a funkció az új HTML5-ös portálon a könyv írásának pillanatában még nem érhető el.

1. Lépjünk be az Azure Management Portálra! A feladat elvégzéshez már rendelkeznünk kell egy SQL Azure-os adatbázissal. (A korábbi részében ennek létrehozását részletesen bemutattuk.)

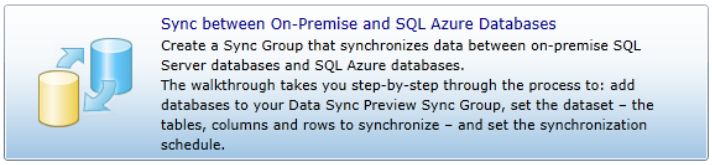
2. Kattintsunk a jobb oldali navigációs sávon a Data Sync menüpontra!

3. A jobb oldalon megjelenő elfőzetések között válasszuk ki azt, amelyikhez kötni szeretnénk. (amihez az SQL Azure adatbázisunk tartozik), majd kattintsunk a Provision gombra!

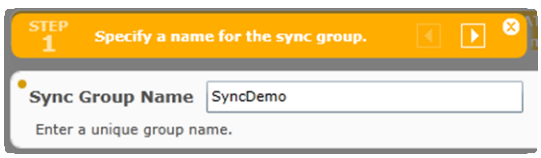
4. Megjelennek a szerződési feltételek. Olvassuk el, majd fogadjuk el őket!

5. A megjelenő ablakban válasszuk ki a számunkra megfelelő régiót. Fontos, hogy a Sync régiójának meg kell egyeznie az adatbázis régiójával. Tehát ha az adatbázisunkat North Europe-ban hoztuk létre, akkor a Sync Regió beállításnak ezzel meg kell egyeznie. Ha ezzel megvagyunk, kattintsunk a Finish gombra!

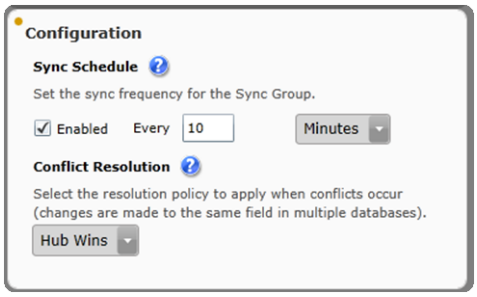
6. Létrehoztunk egy Sync group-ot a kiválasztott régióban! Itt kell kiválasztanunk, hogy on-premise SQL Server és SQL Azure között szeretnénk szinkronizálni vagy SQL Azure adatbázisok között. Kattintsunk a **Sync between On-Premise and SQL Azure Databases** menüpontra!



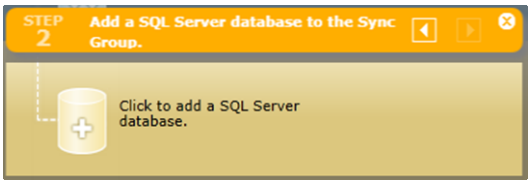
7. A megjelenő ablakban adjuk meg a Sync Group nevét! Ennek a névnek egyedinek kell lennie! Mi most a SyncDemo nevet adjuk neki.



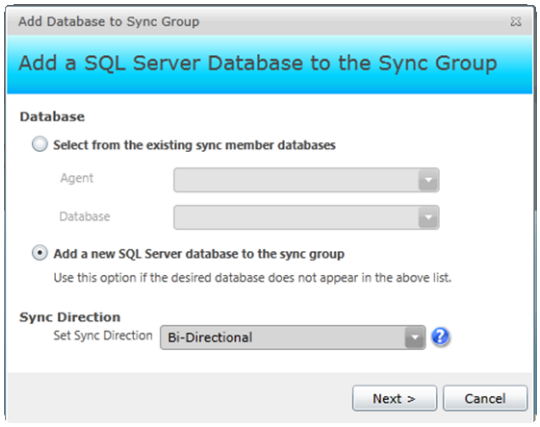
8. A jobb oldalon megjelenő **Configuration** vezérlőnél beállíthatjuk, hogy a szinkronizáció milyen időközönként történjen meg. Ezt mi most 10 percre állítjuk. A legkisebb beállítható szinkronizációs idő 5 perc, a legnagyobb a maximum 1 hónap lehet. Itt állíthatjuk be azt is, hogy konfliktus esetén a kliens (Helyi SQL Sever) vagy a HUB (SQL Azure) nyerjen. Ez a beállítás projektfüggő, de ennél a példánál mi a Hub-ot részesítjük előnyben.



9. Ha beállítottuk a nekünk megfelelő beállításokat, kattintsunk a következő lépés gombra! Második lépésként meg kell adnunk a helyi SQL Server beállításait, amihez szinkronizálni szeretnénk. Kattintsunk a **Click to add a SQL Server database** menüpontra!



10. Megjelenik az **Add Database to Sync Group** ablak. Az ablak alsó részében van a Sync Direction beállítás. Itt állíthatjuk be a szinkronizáció irányát. Megadhatjuk, hogy csak a Hubra szinkronizáljunk (*Sync to the Hub*), megadhatjuk, hogy csak a Hubról szinkronizáljunk a kliens felé (*Sync fron the Hub*) vagy pedig azt, hogy a szinkronizáció kétirányú legyen (Bi-directional). Mi most a kétirányú szinkronizációt fogjuk választani. Konfliktus esetén a Configuration vezérlőnél megadott szabályok érvényesülnek. A Database elemen belül válasszuk ki az **Add a new SQL Server database to the sync group** menüpontot! Ez akkor kell, ha ezelőtt még nem adtunk hozzá ilyen csoportot a Sync Grouphoz. Ellenkező esetben akár ki is választhatnánk a felajánlott listából (*Select from the existing sync member databases*). Ha kiválasztottuk a nekünk megfelelő menüpontot, kattintsunk a Next/Finish gombra!



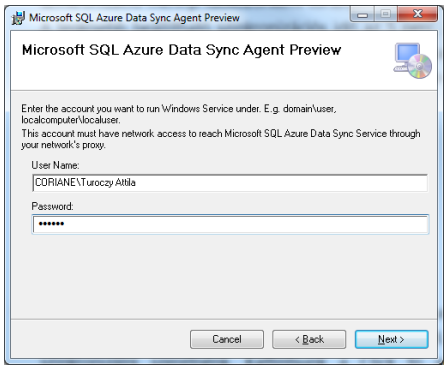
11. A következő ablakban meg kell határoznunk, hogy van-e már Client Sync Agentünk vagy sem. Amennyiben először használjuk ezt a Sync Groupot, nem lesz. Kattintsunk az **Install a new Agent** –re, majd kattintsunk a Next gombra!

Ez az Agent egy kis alkalmazás, ami a szinkronizációt végzi a helyi SQL Server és az Azure között. Ez az alkalmazás http protokollon keresztül kommunikál a felhővel. Így nem szükséges semmilyen tűzfa beállítást módosítani a számára.

Ha a böngészőből elérjük ezt az oldalt, akkor az Agentnek is működnie kell.

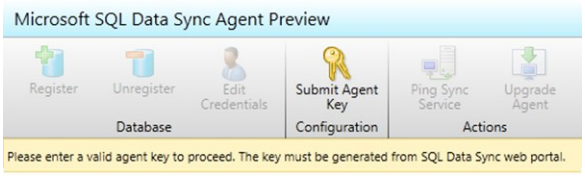
12. A megjelenő ablakban első lépésként töltsük le az SQL Azure Data Sync Agent-et! (Jelenleg Preview változat érhető el!) Ezt az alábbi címről is elérhetjük http://go.microsoft.com/fwlink/?LinkID=226849.

13. Miután letöltöttük, telepítsük az alkalmazást! Az alkalmazás telepítése egyszerű, mindössze egy fontos kérdést tesz fel, mégpedig azt, hogy mi legyen annak a felhasználónak a neve és jelszava, akinek a nevében a szinkronizációs szolgáltatás működni fog! Ennek a felhasználónak el kell tudni érni az SQL Servert is!

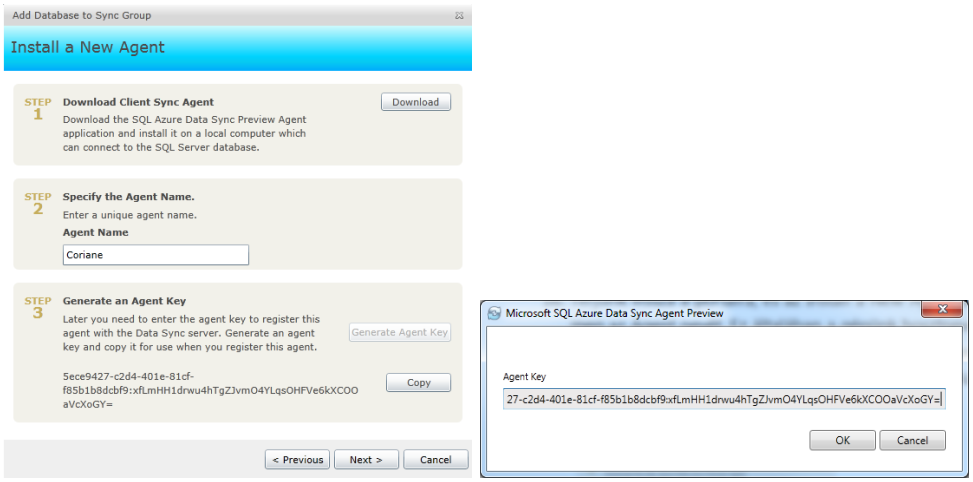


14. Ha sikerült a telepítés, indítsuk el a **Microsoft SQL Azure Data Sync Agent** –et! Ehhez rendszergazdai jogosultságok kellenek!

15. Az alkalmazás elindulását követően kattintsunk a **Submit Agent Key** menüpontra! A megjelenő ablakban egy Agent Key-t vár. Ezt az Agent Key-t a Windows Azure Management portálon találjuk.

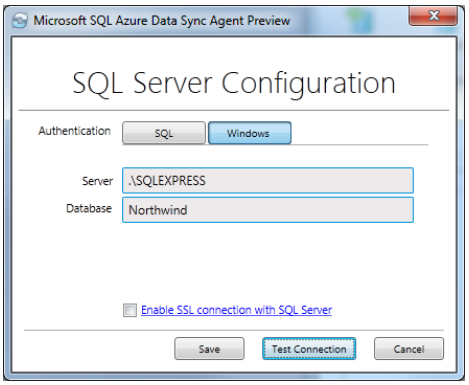


16. Térjünk vissza a portálra, és az **Install a New Agent** ablakban a Step 2 -nél határozzuk meg az Agent nevét! Ez általában a gépünk hosztneve szokott lenni, de lehet más is. Majd kattintsunk a Generate Agent Key menüpontra! Ha sikerült legenerálnia, másoljuk ki a kulcsot, és illesszük be a **Microsoft SQL Azure Data Sync Agent**-be, majd kattintsunk az OK gombra!

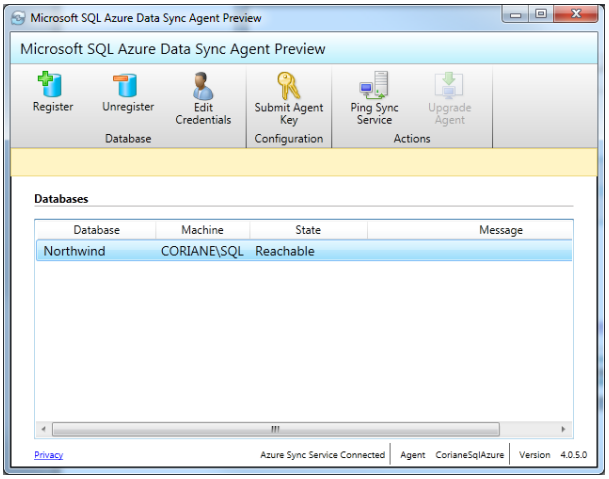


17. Ha ezzel megvagyunk, kattintsunk a **Register** menüpontra! Ekkor felugrik az SQL Server Configuration ablak. Itt adhatjuk meg az SQL Serverünk elérhetőségét. A Servernél az SQL Szerver címét és szükség esetén példányát kell megadnunk. Ez jelen esetben a .\SQLEXPRESS, azaz a lokális gép SQLEXPRESS példánya. A Database-nél azt az adatbázist kell megadnunk, amit szinkronizálni szeretnénk. Ebben a példában ez a

Northwind. Ha Windows Authentication-t választunk, nem kell hitelesítési információkat megadni. SQL Authentikációnál az adott SQL Serverhez és/vagy adatbázishoz tartozó hitelesítési adatokat kell megadnunk.



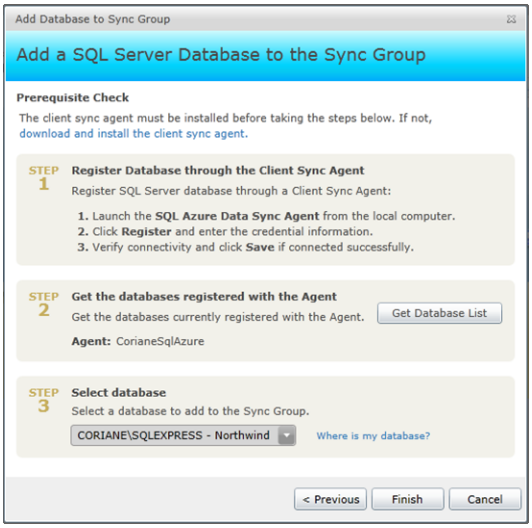
18. Kattintsunk a **Test Connection** menüpontra! Ha sikerül kapcsolódnia, akkor kattintsunk a **Save** gombra! Ekkor a felhasználói felületen megjelenik az az adatbázis, amit előzőleg hozzáadtunk.



19. Térjünk vissza a Management portalra, és az **Install a New Agent** ablakban kattintsunk a **Next** gombra!

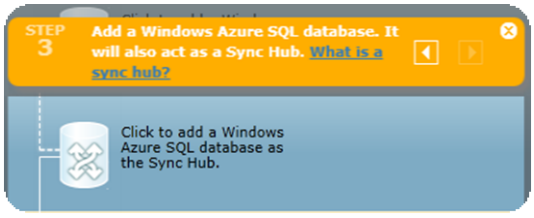
20. Az **Add Database to Sync Group** ablakban kattintsunk a **Get Database List** gombra! Ekkor letölti az összes adatbázis fejlécet, amit a kliensben hozzáadtunk. Jelen esetben csak a Northwindes adatbázis fog megjelenni a lenyíló listában.

21. Válasszuk ki a számunkra szükséges adatbázist! (Jelen esetben azt az egyet, amit előzőleg hozzáadtunk.)

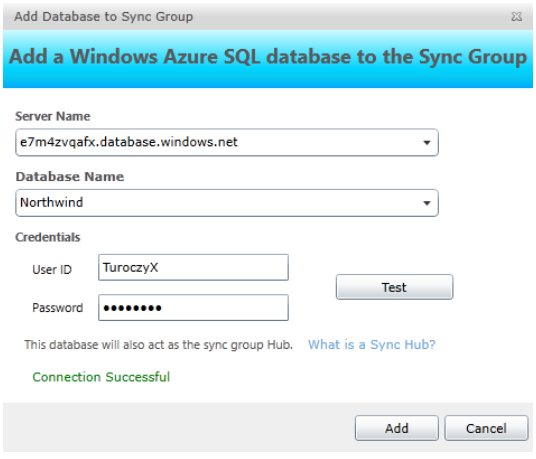


22. Ha ezzel megvagyunk, akkor kattintsunk a **Finish** gombra!

23. Meghatároztuk a kliens gépünk beállításait. Most már csak a cél SQL Azure szerver beállításait kell megadnunk. Kattintsunk a **Click to add a Windows Azure SQL databases as the Sync Hub** gombra!



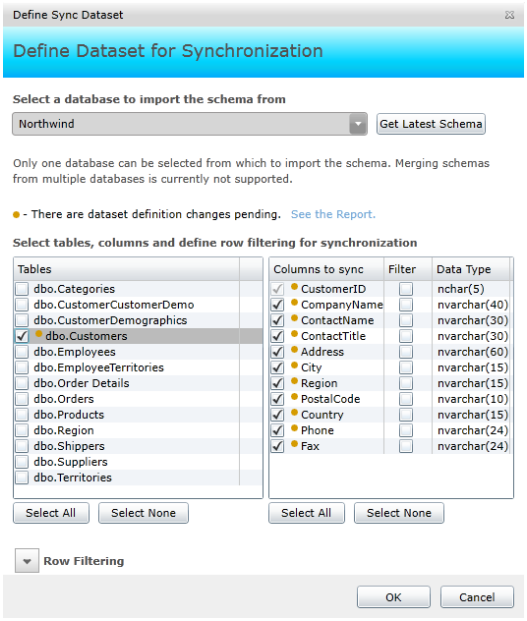
24. A megjelenő ablakban meg kell határoznunk az SQL Azure szerverünk elérhetőségi adatait. Ezt követően kattintsunk az **Add** gombra!



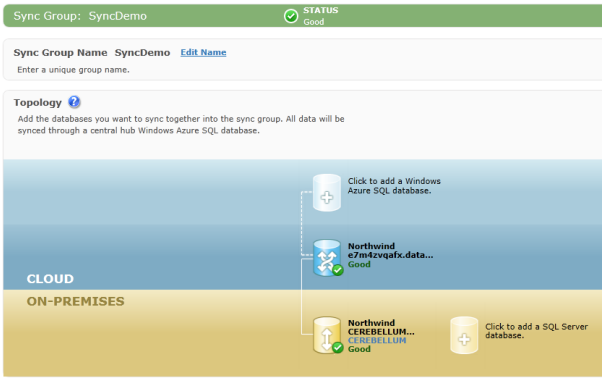
25. A varázsló ekkor ismét a konfliktuskezelést kérdezi meg tőlünk. Ugyanazt, amit korábban már meghatároztunk. Ha nem akarunk ezen változtatni, kattintsunk a tovább szimbólumra!

26. Elérkeztünk az egyik legérdekesebb részhez, a Dataset kezeléshez. A megjelenő ablakban kattintsunk az **Edit Dataset** gombra!

27. Felugrik a Define Dataset for Syncronization ablak. Itt határozhatjuk meg, hogy mely táblákat szeretnénk szinkronizálni. Mi most válasszuk ki a Customers táblát! Azt is meghatározhatjuk, hogy melyik oszlopot szinkronizáljuk. Majd kattintsunk az **OK** gombra!



28. Amint visszatértünk a fő oldalra, láthatjuk az összegzést. Majd kattintsunk a **Deploy** gombra!



Most már minden lépést elvégeztünk. Ha most változtatunk valamit a helyi SQL Serveren, akkor az a következő frissítéstől már az SQL Azure-ban is látszani fog. Abban az esetben, ha az SQL Azure adatbázisban történik változás, a lokális szerver is megkapja az új adatokat a következő frissítéskor. Ne feledjük, most a frissítés kétirányú (bi-directional)! Ha pedig ütközés van, a beállításaink szerint a HUB azaz az SQL Azure adatbázis fog nyerni.