Microsoft SQL Server 2000 Transact SQL – 1. rész
Egyszerû lekérdezések

Bevezetés

Az SQL Server 2000 a 2000. évi adatbázispiac leghangosabban debütáló terméke. A konkurencia is árgus szemekkel figyeli, mit tud az új termék, és igyekszik, hogy elvegye tôle a leggyorsabb adatbázis-kiszolgálónak járó díjat. Nem véletlenül! Már az SQL Server 7 is nagyon kifinomult és hatékony relációs adatbázis-kezelô volt, azonban az utód még igen sok és rendkívül érdekes szolgáltatással rukkolt elô. Ezek kiaknázásához azonban elengedhetetlen a Transact SQL ismerete.
A Transact SQL egy nagyon bonyolult dolog. Legalábbis legtöbben ezzel nyugtatják magukat, amiért nem tanulják meg. Ebben a cikksorozatban rácáfolok erre az állításra. Nagyon kifinomult nyelv, azonban egyszerûbb lekérdezéseket és adatmódosító scripteket bárki képes lesz írni, amint elolvasta és végiggyakorolta ezt a cikksorozatot. Vágjunk hát bele!

Elôkészületek

Adott az SQL 2000 Serverünk. Van rajta egy Northwind nevû adatbázis, ami a szerencséseknek a kiszolgálóval együtt települt fel. *(Aki kevésbé az, az futtassa le az instnwnd.sql scriptet a Program Files\Microsoft SQL Server\Install mappából. Ez létrehozza az adatbázist, és feltölti adatokkal.)* Az adatbázis a Northwind Traders nevû ételkereskedelemmel foglalkozó cég ügyfeleit, eladásait és sok egyéb céges adatát rögzíti. Van benne egy Customers tábla, ami a cég ügyfeleit tartja nyilván, egy Products tábla, ami a kínált termékeket, egy Orders tábla, ami a megrendeléseket, egy Employees, ami az alkalmazottakat. Ezeken kívül van még sok egyéb tábla is benne, de az egyszerûség kedvéért most csak ezeket fogjuk használni.
Adjuk ki az elsô SQL parancsunkat! Nyissuk meg a Query Analyzer nevû programot a Start Menu/Microsoft SQL Server alatt. Ez egyike azon alkalmazásoknak, amelyet nagyon szeretünk használni. Ahogyan a Windows 2000 parancsokat a parancssorban próbálhatjuk ki, hasonló módon a Query Analyzer segítségével lehet tesztelni az SQL lekérdezéseinket. Elindult a program, és kérdezi, hogy melyik kiszolgálóra szeretnénk csatlakozni. Írjuk be gépünk nevét. Ha a lokális gépen van, akkor írjunk *(local)*-t. Ha mi vagyunk a nagyfiúk a gépen *(administrator)*, akkor használjunk Windows NT authentication-t. Ha nem, akkor használjunk SQL authentication-t, sa Login nevet és a megfelelô, leggyakrabban üres jelszót. *(Éles környezetben nem árt ám adni neki egy jelszót! Az esetek igen jelentôs részében elfeledkeznek errôl, és késôbb csodálkoznak, hogy az SQL Servert is futtató webkiszolgálónkon reggelre kicserélték a fôoldalt. Az sa login-nal az SQL Serverre belépve ugyanis kb. 30 mp. alatt rendszergazdává válhat bárki.)* Akik több SQL Servert is futtatnak ugyanazon a gépen, a gépnév\példány nevet írják be a név mezôbe.

Bemelegítés

Sikerült csatlakozni? Akkor a nehezén már túl vagyunk. A Toolbar közepén van egy legördülô lista, ott válasszuk ki a Northwind adatbázist, jelezve, hogy SQL parancsainkat abban szeretnénk végrehajtani. Adjunk a gépnek végre munkát! Kérjük meg, hogy szíveskedjen kilistázni a cég összes vásárlójának a nevét. Ez SQL-ül valahogy így néz ki:

```
SELECT
  CompanyName
FROM
  Customers
```

Írjuk be a parancsot a Query ablakba, majd nyomjunk F5-öt a futtatáshoz. Felszállt a mérôfüst? Ha minden rendben ment, akkor az alsó ablakban 91 céget láthatunk a világ minden részérôl, élen a közismert Alfreds Futterkiste céggel:

```
Alfreds Futterkiste
Ana Trujillo Emparedados y helados
Antonio Moreno Taquería
...
```

Hogyan fordíthatnánk le magyarra az elôbbi SQL parancsot? Hát körülbelül úgy, mintha angolról fordítanánk. Válaszd ki a cégnevet a Customers táblából. Ez a szépsége az SQL nyelvnek, hogy nem For, Loop és egyéb csúfságokkal kell bíbelôdni, hanem majdnem angolul megfogalmazzuk a kérésünket, és az SQL Server végrehajtja azt. A SELECT kulcsszó után megadjuk azoknak az oszlopoknak a nevét, amelyeket szeretnénk látni a parancs kimeneteként, a FROM után pedig azoknak a tábláknak a nevét, ahonnan származnak az adatok. Ebben a példában csak egy forrástáblánk volt.
Néhány gondolat a parancsok tagolásáról. Mint minden programozási nyelvben, sokféle tagolási lehetôség kínálkozik a számunkra *(C programozók szoktak vérre menôen harcolni a kapcsos zárójelek helyérôl)*. Az SQL parancsok akkor lesznek a legolvashatóbbak, ha minden kulcsszót külön sorba írunk, azaz a SELECT is kap egy saját sort, a WHERE is és a többiek is. A kulcsszavakhoz kapcsolódó egyéb paramétereket is célszerû új sorba írni, valamelyest beljebb tolva. A cikk további részében ehhez a konvencióhoz tartom magam.

Feltételek, és ahol a problémák kezdôdnek: a NULL

Fûszerezzük meg a példánkat! Válogassuk le csak a német vásárlókat, azaz akiknek a Country mezôjében Germany áll. Mi sem egyszerûbb:

```
SELECT
  *
FROM
  Customers
WHERE
  Country = 'Germany'

ALFKI Alfreds Futterkiste Maria Anders
Sales Representative      Obere Str. 57
Berlin       NULL   12209  Germany
030-0074321  030-0076545
BLAUS Blauer See Delikatessen    H a n n a
Moos  Sales Representative
Forsterstr. 57     Mannheim      NULL
68306 Germany      0621-08460    0 6 2 1 -
08924
DRACD Drachenblut Delikatessen   S v e n
Ottlieb      Order Administrator
Walserweg 21Aachen NULL   52066
Germany      0241-039123   0241-059428
...
```

Hogy került oda az a csillag? Ez a lusta SQL programozók egyik legnagyobb öröme *(és a Query Optimizer egyik legnagyobb bánata)*. Ha nem akarunk minden oszlopot kézzel felsorolni a SELECT után, akkor használhatjuk a csillagot, ami kilistáz minden oszlopot. Csak bánjunk vele csínján! Gyorsak ugyan a mai számítógépek, de amikor tízezres nagyságrendû sorok mellett egyenként is kilobájtos méretû sorokat válogatunk le, hát bizony bámulhatjuk a homokórát, vagy az idôtúllépésrôl szóló hibaüzeneteket, nem is beszélve a haragos ügyfélrôl.
A WHERE záradék után megadott logikai feltételekkel szûkíthetjük a listázás eredményhalmazát *(Result Set)*. Mi csak azokat a sorokat listáztuk, amelyekben a Country attribútum *(oszlop)* értéke megegyezik a „Germany” szöveggel. Itt tetszôlegesen bonyolult logikai kifejezések állhatnak, csak igyekezzünk jól tagolni, hogy pár hónap múlva is olvasható és érthetô legyen a korábban zseniálisnak kikiáltott feltételünk. Például:

```
SELECT
  CompanyName, Country, Fax
FROM
  Customers
WHERE
  Country = 'Germany' AND
  Fax IS NULL

Königlich Essen     Germany       NULL
Morgenstern Ge...   Germany       NULL
QUICK-Stop          Germany       NULL
```

Azaz kérünk minden olyan sort, ahol a cégnév Germany, és a Fax mezô értéke NULL. De mi az a NULL? Ezt a kérdést sokszor több éves tapasztalattal rendelkezô SQL programozók sem tudják, pedig igen lényeges a tisztánlátás az ô tulajdonságait illetôen.
A NULL egy SQL adatbázisban azt jelenti, hogy nincs adat. Nem keverendô össze a 0 számmal, vagy az „” üres sztringgel, még kevésbé a NULL pointerrel! Majd az illesztéseknél *(JOIN)* látjuk, hogy igen sok problémát okozhatnak, úgyhogy szemmel kell tartani ôket. Látható, hogy az összehasonlító operátor *(is)* is más, mint amit egyéb, „normális” adatra alkalmazunk (=, <, > stb.). A nyelv szerzôi ezzel is meg akarták különböztetni a NULL-t a valós adatoktól. Hisz 2 > NULL az igaz? Hmm. A jó ég tudja. Pont ezért problémás a NULL adatok kezelése, mert amíg normális adatokra megfelelôen mûködnek az operátorok, NULL-ra nem. Ez nem jelenti azt, hogy hibát jeleznek, vagy véletlenszerûen viselkednek. Természetesen következetes módon fognak viselkedni, csak meg kell nézni a kézikönyvben, hogy milyen feltételek esetén milyen eredmény adnak, vagy ami ennél százszor jobb megoldás: ahol csak lehet, kerüljük el a NULL-okat. Ínyenceknek: a Microsoft SQL Server fizikai adattárolási struktúrája olyan, hogy minden egyes NULL engedélyezett attribútumhoz van egy jelzôbit, amely jelzi, hogy tartalmaz-e a mezô tényleges adatot, vagy *nem (azaz NULL)*. Ez a sort leíró fejlécben van, hisz ezt minden sor minden NULL-ozható attribútumára tárolni kell. Azaz ez a plusz adminisztráció még egy kis teljesítményhátrányt is jelent.

Csak számolunk, csak számolunk

Hagyjuk a ronda NULL-okat, és haladjunk tovább a normális adatok világában. Mi van, ha a big boss abc sorrendben szeretné látni az amerikai megrendelôket?

```
SELECT
  CompanyName
FROM
  Customers
WHERE
  Country = 'USA'
ORDER BY
  CompanyName

Great Lakes Food Market
Hungry Coyote Import Store
Lazy K Kountry Store
Let's Stop N Shop
...
```

F5, és nagy az öröm. Az ORDER BY után több mezôt is fel lehet sorolni, azaz, hogy elôször az elsô mezô szerint rendezzen sorba, majd ezeken a csoportokon belül rendezze tovább a második, stb. attribútum szerint. A rendezô mezônevek közé vesszôt kell rakni, hasonlóan, mint a SELECT után. Pl.:

```
SELECT
  Country, City, CompanyName,
ContactName
FROM
  Customers
ORDER BY
  Country, City, CompanyName
```

A fônök kíváncsisága csillapíthatatlan. Kíváncsi rá, hogy meddig ér a lepedôje, azaz mely országokkal van kapcsolata a cégének. Ám legyen:

```
SELECT DISTINCT
  Country
FROM
  Customers

Argentina
Austria
Belgium
Brazil
...
```

Mit is mondtunk az SQL Servernek? Listázza ki az összes *különbözô (DISTINCT)* országot a Customers táblából Ez olyan egyszerû SQL-ben, mint ahogyan látszik! Aki hozzászokott a procedurális gondolkodáshoz, annak szokatlan ez a fajta gondolkodásmód, ahogyan SQL nyelven definiáljuk a problémákat. Egy procedurális nyelven *(C, Pascal, Basic stb.)* úgy válogatnák le a sorokat, hogy végigmennénk minden soron, egybegyûjtenénk a már megtalált országokat, és minden egyes még feldolgozatlan sornál megnéznénk, hogy már benne van-e az adott tétel a megtaláltak között. Ha nem, felvesszük, ha igen, akkor csak egyszerûen továbblépünk a következôre. Azaz procedurális esetben sorokban, egyedi adatokban gondolkodunk, míg a másik esetben halmazokban. Több év C által megfertôzött szekvenciális gondolkodás után szokatlan lehet ez, de köny-

nyen meg lehet szokni, és miután valaki ráérzett az ízére, rájön, hogy ilyen módon sokkal egyszerûbben és tömörebben meg lehet fogalmazni a problémákat *(nem véletlen, hogy ily módon fejlesztették ki az SQL nyelvet)*. Másrészt kurzorok használatában át lehet térni szekvenciális feldolgozásra, de errôl majd egy teljes cikk fog szólni.
Na, de félre az ideológiákkal, és számoljuk meg, hány ügyfelünk van Franciaországban:

```
SELECT
  COUNT(*)
FROM
  Customers
WHERE
  Country = 'France'

11
```

Magyarul: számolja meg az összes sort, ahol az ország oszlopban Franciaország áll. A COUNT az aggregáló függvények egyik jeles képviselôje. Ezeknek a függvényeknek az a közös jellemzôje, hogy több sorból képeznek valamilyen végeredményt, a sorok valamelyik attribútumát felhasználva. Ebbôl a szempontból COUNT(*) egy kicsit speciális, mert az oszlopoktól függetlenül egyszerûen megszámolja a sorokat. Egy újabb példán keresztül nézzünk egy kicsit mögé a COUNT függvénynek.
Hány cégnek van faxa, vagy legalábbis úgy tudjuk róla, hogy van neki (azaz a Fax mezô nem NULL)?
Klasszikus megoldás:

```
SELECT
  COUNT(*)
FROM
  Customers
WHERE
  Fax IS NOT NULL

69
```

Microsoft SQL Server specifikus, de jól mûködô megoldás:

```
SELECT
  COUNT(Fax)
FROM
  Customers

69
```

Miért mûködik ez jól? Miért nem számolja meg az összes sort, miért csak azokat, amelyekben a Fax mezô nem NULL? Mert így logikus. Mivel a NULL azt jelenti, hogy nincs adat, vagy nem tudunk róla semmit, ezért nem is szabad belevenni az ilyen számlálásokba. Ez megint csak a NULL értékek sajátossága *(mondtam, hogy sok baj van a NULL-okkal)*. Az összes aggregáló függvény bokorugorásban megy tovább a következô sorra, ha a feldolgozás alatt álló mezôben NULL van. Hisz mit kezdene egy átlagszámító függvény a NULL-lal?
A COUNT(*) azért lóg ki a sorból, mert minden sort beszámol, még akkor is, ha minden attribútum értéke NULL. A két megoldás egyformán hatékony, de míg az elôbbi jól olvasható és nem tartalmaz implicit megállapodásokat, addig a második néhány hónappal a fejlesztés után már fejtörést okozhat, hogy mi a csudát akartam én kezdeni azzal a kifejezéssel. Azaz nem javaslom a második megoldást. Aki úgy érzi, hogy a második megoldás gyorsabb, annak elmondom, hogy az SQL Server esetén szintaktikai bravúrokkal általában nem sikerül teljesítményt javítani, mert úgyis azzal kezdi az utasítások végrehajtását, hogy lebontja ôket elemi részekre, és az *(általa vélt)* legoptimálisabb módon fogja végrehajtani. Eddig még legtöbbször okosabb volt nálam :)
Maradjunk még egy kicsit az aggregáló függvényeknél! További tipikus példák a MIN, a MAX és az AVG és a SUM. Az elsô kettô egyértelmû, az AVG számtani közepet *(népi nyelven átlag)* számol, a SUM pedig összeadja a megadott mezôket. Nézzünk egy példát, amiben felhasználjuk mindegyiket! Keressük meg a legkisebb és a legnagyobb egységárú termék árát, az egységárak átlagát és az egységárak összegét az összes termékre vonatkozóan.

```
SELECT
  MIN(UnitPrice),
  MAX(UnitPrice),
  AVG(UnitPrice),
  SUM(UnitPrice)
FROM
  Products

2.5000     263.5000     28.8663     2222.7100
```

Veszedelmes viszonyok

Vegyünk nagy levegôt, és mélyedjünk el egy kicsit az adatbázis-kezelés elméletébe. Miért mondják az SQL Serverre, hogy relációs adatbázis-kezelô? Azért, mert a benne levô entitások *(ennek magyar megfelelôje az izé, de fizikai adatbázisban táblának felel meg)* között logikai kapcsolatokat, relációkat fogalmazhatunk meg. Ezt azért találták ki, mert így az ismétlôdô adatokat kiemelhetjük külön táblákba, egyrészt helyet spórolva meg, másrész az adatbázis konzisztenciájának biztosítása végett. Lássunk erre egy példát. Az Orders tábla tárolja az ügyfelek vásárlásait. Azonban minden egyes sorban az ügyfelet csak egy azonosító jelzi *(pl.: RATTC)*, amely azonosító alatt futó ügyfél valódi adatai a Customers táblában vannak. Ezzel helyet spóroltak, hisz nem kell többször leírni az ügyfél címét, nevét stb. minden egyes megrendelésnél, másrészt nem fogunk találni olyan sorokat, hogy megrendelô Egér Béla, a másikban, hogy Eger Bela és így tovább. Így az adatbázis logikailag konzisztens marad, hisz ugyanarra a valóságos egyedpéldányra *(Éger Béla)* nem hivatkozhatunk többféleképpen. Egyébként azt a folyamatot, amikor a redundáns részeket kirakjuk külön táblába, normalizálásnak nevezzük. Ebbôl következôen azok az adatbázisok normálisak, amelyekben sok, de keskeny, azaz kevés oszlopot tartalmazó tábla van. Az elméleti maximumig agyonnormalizált adatbázist azonban keveset látunk a világban. Ennek két oka van. Egyrészt sokan nem is tudják, hogyan kell normalizálni *(és hogy egyáltalán kell)*, másrészt teljesítmény-meggondolások miatt sokszor normalizálás után denormalizáljuk valamelyest az adatbázisunkat. Egy nem megfelelôen normalizált adatbázissal az a baj, hogy nehéz a konzisztenciáját megtartani. Például egy oszlopban tároljuk egy termék árát, egy másikban az ÁFÁ-ját, egy harmadikban pedig az ÁFÁ-s árat. Ez gyönyörû példája a NEM normalizált adatbázisnak *(triviális függôség van az oszlopok között)*. A baj akkor kezdôdik, amikor megváltozik a termék ára, és elfelejtjük módosítani az ÁFÁ-s árat. Egy hét *(perc)* múlva, amikor már senki nem emlékszik, melyik mezô volt módosítva, melyik árat fogadjuk el helyesnek?

Újra együtt!

Vissza a gyakorlatba, hogyan lehet összehozni a szétdarabolt információkat? Ehhez lesz szükségünk a JOIN kulcsszóra. Listázzuk ki a megrendelôk által kért megrendelések dátumát és a szállítás városát:

```
SELECT
  CompanyName, OrderDate, ShipCity
FROM
  Customers
INNER JOIN
  Orders
ON
  Customers.CustomerID =
Orders.CustomerID

CompanyName OrderDate ShipCity
Vins et alcools Chevalier 1996-07-04
Reims
Toms Spezialitäten 1996-07-05 Münster
Hanari Carnes 1996-07-08 Rio de Janeiro
...
```

Azaz a Customers táblát illessze az Orders táblához, mégpedig azokon a pontokon, ahol a Customers tábla CustomerID mezôje megegyezik az Orders tábla CustomerID mezôjével. Minden egyes egyezésnél készít egy „hosszú” sort, azaz egymás mellé rakja a két tábla összeillesztett sorát, amelyekbôl mi csak a SELECT után felsorolt oszlopokat kérjük. SQL Server 6.5-ig ez úgy ment, hogy vette az elsô tábla elsô sorát, és megnézte, hogy az ON után megadott feltétel alkalmazásával a másik táblában talál-e egy párt a vizsgált sornak. Ha igen, akkor egymás mellé illesztette ôket, letárolta, és folytatta a második tábla következô sorával, hisz általában több sor is illeszkedhet a vizsgált sorhoz. Ez elsô ránézésre hihetetlen lassú folyamat, hisz a két tábla sorai számának szorzata adja az összehasonító mûveletek számát. Az indexek használata miatt ez szerencsére nem így van. Ennek az illesztési eljárásnak a *(jól eltalált)* neve Nested Loop Join. SQL 7-tôl még két további illesztô algoritmus áll rendelkezésre, amelyeket nagyon sok sort adó lekérdezéseknél szeret választani *(Hash és Merge Join)*.
A feladat jellegének megfelelôen tetszôleges számú táblát össze lehet kapcsolni. Pl.:

```
SELECT
  CompanyName, OrderDate,
  ProductName, Products.ProductID
FROM
  Customers
INNER JOIN
  Orders
ON
  Customers.CustomerID =
Orders.CustomerID
INNER JOIN
  [Order Details]
ON
  Orders.OrderID = [Order
Details].OrderID
INNER JOIN
  Products
ON [Order Details].ProductID =
Products.ProductID

CompanyName OrderDate  ProdName
ProdID
Blondesd... 1996-07-25 Alice Mutton 17
Lehmanns... 1996-08-13 Alice Mutton 17
Rattlesn... 1996-08-30 Alice Mutton 17
```

Két dolgot figyeljünk meg a fenti lekérdezésben! Ha olyan oszlopot listázunk ki, amelynek neve több táblában is szerepel *(pl.: ProductID, OrderID, CustomerID)*, akkor meg kell jelölni, hogy melyik táblából akarjuk kilistázni az adatokat. Ez INNER JOIN-nál még mindegy lenne, de a többi JOIN-nál nagyon fontos lesz. A másik lényeges pont, hogy a szóközt is tartalmazó tábla- és oszlopneveket szögletes zárójelek [] közé kell tenni. Ugyanez érvényes, ha kulcsszót akarunk használni tábla- vagy attribútumnévnek. Where nevû tábla nem túl gyakori, de a User tábla már annál inkább, ami pedig kulcsszó...
Az INNER JOIN mûködésébôl következik, hogy az összes olyan sor kimarad az eredménybôl, amelynek nincs párja a másik táblában. Azonban a gyakorlatban sokszor az árva sorokra is szükség van. A kettôvel ezelôtti példánál maradva azokat a vásárlókat is ki szeretnénk listázni, akiknek *(még)* nincsenek megrendeléseik. Ekkor jön segítségünkre az OUTER JOIN. Ennek két alfaja van, a LEFT OUTER JOIN és a RIGHT OUTER JOIN. Mivel a JOIN kulcsszó mindig két tábla között helyezkedik el, a jobb és bal irány értelmezése természetesen adódik. Amelyik táblát így kitüntetjük, abból az összes sor kilistázódik, azok is, amelyeknek nincs párja a másik oldali táblában. Így a példánk:

```
SELECT
  CompanyName, OrderDate, ShipCity
FROM
  Customers
LEFT OUTER JOIN
  Orders
ON
  Customers.CustomerID = Orders.CustomerID

Rattlesnake Canyon 1998-05-06 Albuquerque
Paris spécialités  NULL        NULL
FISSA Fabrica      NULL        NULL
```

Ahol az ügyfélhez nincs megrendelés az Orders táblában, ott a kiszolgáló NULL-t helyez el azokban az oszlopokban, amelyek az Orders táblára mutatnak. Ez egybevág a NULL azon jelentésével, hogy „nincs adat”.
Sajnos ezen a ponton be kell fejezzem barangolásomat a lekérdezések birodalmában, de ne aggódjanak, a következô számban tovább túrunk-fúrunk a SELECT rejtelmeiben, hogy azután rátérhessünk az igazi izgalmakra, az adatok módosítására.

Soczó Zsolt, MCSE
Protomix Kft.

Transact SQL 2. rész

Összetett lekérdezések

Cikkünk elôzô részében elindultunk a Microsoft SQL 2000 programozásának izgalmas útján. Megnéztük, hogyan írhatunk egyszerû lekérdezéseket, amelyekkel apróbb feladatokat adhatunk az adatbázisnak. Ebben a részben jobban belemélyedünk a lekérdezések lelkivilágába, és megnézzük, hogy komolyabb feladatokat hogyan oldhatunk meg a Transact SQL segítségével.

Még mindig együtt!

Elôzô cikkünk végén az illesztésekkel *(JOIN)* foglalkoztunk, és eljutottunk odáig, hogy illesztés segítségével logikailag összetartozó, de fizikailag több táblára szétdarabolt adatokat újra egyesíthetünk. Megbeszéltük, hogy az INNER JOIN segítségével meg lehet találni az összetartozó sorokat. Megnéztük, hogy vannak olyan esetek, amikor nemcsak a párok érdekesek, hanem szükség van azokra a sorokra is, amelyekhez nincs kapcsolódó sor más táblákban, ilyenkor használtuk az OUTER JOIN-t. A LEFT és a RIGHT OUTER JOIN segítségével kilistáztathattuk azokat a sorokat is, amelyeknek nem volt párja a másikban.

Az OUTER JOIN eddig még nem említett válfaja a FULL OUTER JOIN. Ez mindkét tábla tartalmát kilistázza, függetlenül attól, hogy talált-e egyezést a másik táblában, vagy sem. Ennek felhasználása már elég speciális. Például a Northwind adatbázis Customers és Orders táblái között egy LEFT OUTER JOIN-nak van értelme, hisz kilistázza azokat a vásárlókat, akiknek nincsenek megrendeléseik. A fordított helyzet *(egy jól megtervezett és implementált adatbázisban)* elvileg elô sem állhat, azaz, hogy vannak olyan megrendelések, amelyekhez nincs megrendelô. Ez a hivatkozási *(referential)* integritás megsértése volna, hisz az Orders táblában van egy idegen kulcs *(foreign key)* a Customers táblára. Ennek ellenére a gyakorlatban sokszor elôfordul, fôleg, amikor egy régebbi rendszerbôl költöztetünk adatokat egy újabba, hogy bizony sok helyen baj van az adatok épségével. Tegyük fel, hogy az elôbb említett két táblát egy másik adatbázisból kaptuk, és az a feladatunk, hogy állapítsuk meg, rendben vannak-e a hivatkozási szabályok. Mi sem egyszerûbb:

```
SELECT
   Customers.CustomerID,
   Customers.CompanyName,
   Orders.CustomerID
FROM
   Customers
FULL OUTER JOIN
   Orders
ON
  Customers.CustomerID = Orders.CustomerID
WHERE
   Orders.CustomerID IS NULL OR
   Customers.CustomerID IS NULL
```

Mit várunk a lekérdezéstôl? Azt, hogy kilistázza az összes megrendelést, amelynek nincs gazdája *(Customers.CustomerID IS NULL)*, és kilistázza azokat a vásárlókat, akiknek nincs megrendelése *(Orders.CustomerID IS NULL)*. Az adatok értelmezését figyelembe véve csak az elôbbi valódi probléma, az utóbbi nem. Ez azért van, mert logikailag a Customers és az Orders tábla között egy vagy több kapcsolat van, azaz minden tételhez a Customers táblában tartozhat nulla vagy több tétel a másikban. Ennek megfordításaként viszont minden egyes tételhez az Orders táblában kell lennie egy megfelelô tételnek a Customers táblában. A teljesség kedvéért íme a lekérdezés kimenete, melybôl látszik, hogy nincs árva megrendelés *(ahol az elsô CustomerID NULL értékû lenne)*:

```
CustomerID CompanyName          CustomerID
FISSA      FISSA Fabrica S.A.   NULL
PARIS      Paris spécialités    NULL
```

Az illesztések közül már csak egy maradt hátra, amelyet csak nagyon ritkán, elsôsorban tesztadatok generálására használunk. Ennek neve CROSS JOIN, és a matematikából ismert Descartes szorzatot valósítja meg. Adatbázisra lefordítva ez azt jelenti, hogy az elsô tábla minden sorát összepárosítja a másik tábla minden sorával, azaz tulajdonképpen egy speciális INNER JOIN, amelynek feltétel része *(ON ...)* mindig igaz.

Nézzük meg, hogy a CROSS JOIN segítségével hogyan lehet kevés kiinduló adatból nagyszámú tesztadatot generálni! Tegyük fel, hogy teszt felhasználókra van szükségünk. Kiindulásként felvittük kilenc személy vezeték- és keresztnevét egy táblába *(Employees)*, azt szeretnénk, hogy a vezeték- és keresztnevek kombinálásával elôállítsunk felhasználói neveket. Ha minden vezetéknevet összepárosítunk minden keresztnévvel, akkor 9x9=81 nevet fogunk kapni. Hogyan néz hát ki a generáló script?

```
SELECT
  E1.FirstName, E2.LastName
FROM
  Employees E1
CROSS JOIN
  Employees E2

...
Robert      Buchanan
Laura       Buchanan
Anne        Buchanan
Nancy       Callahan
Andrew      Callahan
...
```

Két csel is el van rejtve ebben a rövid lekérdezésben. Lehet egy táblát önmagával illeszteni? Igen! Ezt hívják self joinnak. De honnan tudja a SELECT, hogy mikor melyik példányra hivatkozunk? Onnan, hogy átnevezzük ôket, álnevet adunk nekik *(alias)*. Így bárhol a lekérdezésben az elsô Employees táblára E1 néven lehet hivatkozni, míg a másikra E2 néven. Így a fordító nem fog kétségek között vergôdni, hogy éppen mire gondoltunk. Tábla álneveket bármikor használhatunk, nem csak illesztések esetén. Sokszor hosszú táblaneveket rövidítünk velük, például az [Orders Details] táblát od-re.

Maradjunk még az illesztéseknél, mert sok olyan finomság van bennük, amelyeket ha nem tud valaki elôre, csak több napos bosszankodás után fogja felfedezni.

A fránya *=

Az illesztések formális leírásának van egy másik formája is, amelyet szándékosan elhanyagoltam eddig, mert elavult, és nem lehet vele minden feladatot egzaktul megfogalmazni. Mivel azonban nagyon sokan használják, nem hagyhatom ki a tárgyalásból, hisz ha már együtt kell élnünk vele, legalább ismerjük az árnyoldalait. Összehasonlításként leírok egy lekérdezést az INNER JOIN felhasználásával, majd a régi módon:

Microsoft BackOffice

```
SELECT
  Customers.CustomerID,
  Customers.CompanyName,
  Orders.OrderDate
FROM
  Customers
INNER JOIN
  Orders
ON
  Customers.CustomerID =
Orders.CustomerID
```

Régi módon:

```
SELECT
  Customers.CustomerID,
  Customers.CompanyName,
  Orders.OrderDate
FROM
  Customers, Orders
WHERE
  Customers.CustomerID =
Orders.CustomerID
```

Azaz válogassa ki azokat a sorokat a két táblából, ahol *(WHERE)* a Customers.CustomerID = Orders.CustomerID. Teljesen logikus, és nincs is vele baj. Ha keresni akarjuk a kákán a csomót *(és miért ne tennénk)*, akkor hogy van az, hogy a WHERE után lehetnek olyan kifejezések, amelyek sorok szûrését végzô feltételeket tartalmaznak, és olyanok is, amelyek táblák logikai összekapcsolását tartalmazzák? Nem két, teljesen különbözô funkcióról van itt szó? De! És ez visszsza is fog ütni mindjárt *(megvan az elsô csomónk)*!
A probléma az OUTER JOIN-oknál kezdôdik. A régebbi OUTER JOIN-t megvalósító kifejezés a WHERE a *= b volt, és attól függôen, hogy a csillag melyik oldalán van az egyenlôségjelnek, lehet jobb vagy bal oldali illesztést kifejezni. Ez ugyanazt jelenti, mint az OUTER JOIN? *(És most mindenki tegye fel és válaszolja meg magának ezt a kérdést, mielôtt tovább olvasna.)* Nem! Nézzük meg egy példán keresztül a csalást!
Új formátum:

```
SELECT
  Customers.CustomerID,
  Customers.CompanyName,
  Orders.OrderDate
FROM
  Customers
LEFT JOIN
  Orders
ON
  Customers.CustomerID =
  Orders.CustomerID
```

Régebbi formátum:

```
SELECT
  Customers.CustomerID,
  Customers.CompanyName,
  Orders.OrderDate
FROM
  Customers, Orders
WHERE
  Customers.CustomerID *= Orders.CustomerID
```

A két lekérdezés kimenete azonos:

```
BONAP       Bon app'            1998-05-06
RATTC       Rattlesnake Canyon  1998-05-06
PARIS       Paris spécialités   NULL
FISSA       FISSA Fabrica S.A.  NULL
```

Akkor miért kritizálom a *= formátumot? Mindjárt kiderül. Próbáljuk meg kiszûrni például a Bon app' céget a listából, ügyelve arra, hogy a LEFT JOIN által behozott NULL-okat ne szûrjük ki. Az új formához csak egy feltételt kell adni:

```
WHERE
  Orders.CustomerID <> 'BONAP' OR
  Orders.CustomerID IS NULL

RATTC       Rattlesnake Canyon  1998-05-06
PARIS       Paris spécialités   NULL
FISSA       FISSA Fabrica S.A.  NULL
```

És a kimenetbôl tényleg eltûnt a kérdéses sor, míg a NULL-osok megmaradtak. Egy kis magyarázatot azért megér, hogy miért kell a második feltétel is, miért nem elég csak az elsô. Az elôzô részben részletesen foglalkoztunk vele, hogy a NULL azt jelenti, hogy nincs adtat, így az Orders.CustomerID <> 'BONAP' feltételnél kiesnének a NULL-okat tartalmazó sorok, mert egy NULL-al végzett összehasonlításnak nem lehet eldönteni az igazságtartalmát. Ezért kellett bevetni az IS NULL-t.
Itt az ideje, hogy kiugrasszuk a nyulat a bokorból! Írjuk át a lekérdezést a régi szintaxisra:

```
WHERE
  (Customers.CustomerID *=
  Orders.CustomerID) AND
  (Orders.CustomerID <> 'BONAP' OR
  Orders.CustomerID IS NULL)
```

És mit látunk kimenetnek?

```
RATTC       Rattlesnake         1998-05-06
PARIS       Paris spécialités   NULL
FISSA       FISSA Fabrica S.A.  NULL
BONAP       Bon app'            NULL
```

Ott virít a Bon app', pedig kiszûrtük *(ráadásul lemaradt a megrendelése is)*! Miért? Azért, mert a NULL ellenôrzése elôbb történik meg a szerverben, mint az illesztés, így a bal illesztés behozza újra a kiszûrni kívánt sort. De hogy lehettünk ennyire figyelmetlenek, miért nem a Customers.CustomerID-ra szûrünk, miért az Orders.CustomerID-ra. Ez biztos bejön! Nézzük csak:

```
WHERE
  (Customers.CustomerID *=
  Orders.CustomerID) AND
  (Customers.CustomerID <> 'BONAP' OR
  Orders.CustomerID IS NULL)
```

Nem mutatom meg a kimenetet, de még mindig benne van a BONAP! Hogy lehet ez? Úgy, hogy a Orders.CustomerID IS NULL most az illesztés után hajtódott végre, így behozta a BONAP-ot. Szeszélyesebb az adatbázis motor, mint az idôjárás! Vagy mégsem?

A megoldás

Ne fokozzunk tovább a feszültséget! Miután rájöttünk, hogy az IS NULL vizsgálat hozza be a nemkívánatos sorokat, vegyük ki a lekérdezésbôl.

```
WHERE
   (Customers.CustomerID *=
Orders.CustomerID) AND
   (Customers.CustomerID <> 'BONAP')
```

És eltûnt a Bon app'! Lehet, hogy ez sok hûhó semmiért, és hogy ez egy olyan nyilvánvaló dolog volt, amit egy tapasztalt SQL programozó azonnal kiszúr. Lehet, bár azért még nekik is lehet fejtörést okozni. Mert rakjuk csak be a régi stílusú külsô illesztésünket egy SQL nézetbe *(View)*. Azoknak a kedves olvasóknak, akik még nem használtak nézeteket, néhány szó róla. Nézetekbe olyan lekérdezéseket szoktunk „becsomagolni“, amelyeket több helyen is fel fogunk használni, így nem kell mindig leírni ôket. A nézetek úgy viselkednek, mintha ôk valamiféle virtuális táblák lennének, amelyek a bennük található lekérdezéseket táblaként adják vissza. Na már most, tegyük fel, hogy több ember dolgozik egy projekten. Az egyik megírja a már sokszor emlegetett lekérdezést egy nézetbe, természetesen a régi szintaxissal. Legyen a nézet definíciója:

```
CREATE VIEW
   CustOrders
AS
SELECT
   Orders.CustomerID,
   Customers.CompanyName,
   Orders.OrderDate
FROM
   Customers, Orders
WHERE
   Customers.CustomerID *=
   Orders.CustomerID
```

Ezután a tudatlanság boldogságában leledzô társprogramozó ki szeretné szûrni a BONAP-ot:

```
SELECT
   CustOrders.CustomerID,
   CustOrders.CompanyName,
   CustOrders.OrderDate
FROM
   CustOrders
WHERE
   CustOrders.CustomerID <> 'BONAP' OR
   CustOrders.CustomerID IS NOT NULL
```

És itt áll égnek a haja, mert a szerver látszólag nem mûködik normálisan, hisz nem szûrte ki a megfelelô sort!

Összegezve: ne használjuk a régi formátumú illesztéseket, mert félreértésekhez vezethet. Emellett a késôbbi verziójú SQL Serverek nem fogják támogatni. Ha más nem, ez elégséges érv lehet.

Egymásba ágyazva

Az SQL nyelv egyik leghatékonyabb eszköze, hogy a WHERE feltételbe nemcsak egyszerû logikai kifejezéseket írhatunk, hanem további lekérdezéseket is. Ráadásul a két lekérdezés között lehet kapcsolatot is teremteni. Például listázzuk ki azokat az alkalmazottakat, akik már teljesítettek megrendeléseket, azaz az Orders táblában van olyan sor, ami az Employee táblában található alkalmazottra mutat:

```
SELECT
   LastName, FirstName
FROM
   Employees
WHERE
   Employees.EmployeeID IN
   (SELECT
      EmployeeID
    FROM
      Orders)
```

Másképpen fogalmazva listázzuk ki az EmployeeID-kat az Orders táblából, majd az Employees táblából válogassuk le azokat a sorokat, amelyeknek az EmloyeeID-ja megegyezik valamelyik Orders táblából származó EmployeeID-val (IN). Ezt a lekérdezést át lehetne írni JOIN-ra is:

```
SELECT
   LastName, FirstName
FROM
   Employees
INNER JOIN
   Orders
ON
   Employees.EmployeeID =
Orders.EmployeeID
```

Általánosságban igaz, hogy minden JOIN-t át lehet írni egymásba ágyazott lekérdezésre, de visszafelé ez nem feltétlenül igaz. Azaz vannak olyan egymásba ágyazott lekérdezések, amelyek egy egyszerû illesztésnél bonyolultabb dolgot valósítanak meg. Például listázzuk ki azokat a termékeket *(Products)*, amelyek ára 10 dollár alatt van, de volt olyan alkalom, amikor egyszerre eladtak valamelyik termékbôl több mint 800 dollárnyit.

```
SELECT
   ProductID, ProductName
FROM
   Products AS p
WHERE
   UnitPrice < 10 AND
   ProductID IN
   (SELECT ProductID
    FROM
      [Order Details] od
    WHERE
      od.Quantity * p.UnitPrice > 800)

ProductID   ProductName
41           Jack's New England Clam
45           Rogede sild
75           Rhönbräu Klosterbier
```

Microsoft BackOffice

Hogyan képzelhetjük el a lekérdezés mûködését? Az elsô SELECT végiglépked a Products tábla azon sorain, melyekben a UnitPrice mezô értéke kisebb, mint 10. Minden kiválasztott sornál elindít egy belsô ciklust *(a belsô SELECT)* az Order Details táblára, és keres olyan sorokat, amelyekre teljesül a WHERE-ben megadott feltétel. Ennek specialitása, hogy a külsô SELECT által pillanatnyilag kiválasztott sorból származó adatot is felhasználja *(pl.UnitPrice)*. Az ilyen típusú egymásba ágyazott lekérdezéseket Correlated Subquery-nek nevezzük.
Mikor érdemes használni egymásba ágyazott lekérdezéseket, és mikor illesztést? Az attól függ. Ha nincs különösebb követelmény a lekérdezés teljesítményére, akkor használjuk azt, ami a probléma természetes nyelvi megfogalmazásához legközelebb áll, így késôbb könnyebben érthetô és karbantartható lesz a script. Ha fontos az optimális teljesítmény, akkor inkább használjunk illesztést. Miért? Illesztések esetén az optimalizáló meg tudja választani, hogy milyen sorrendben hajtsa végre az utasítást, így minimalizálni tudja a végrehajtáshoz szükséges költséget. Egymásba ágyazott lekérdezéssel beledrótozzuk a végrehajtás sorrendjét a lekérdezésbe, így nem sok teret hagyunk az optimalizálónak a gondolkodásra. Ennek ellenére majd mutatok eseteket, amikor lassabb lesz az illesztés. A végsô ítéletet csak a lekérdezés költségének vizsgálatával lehet kimondani *(Query Analyzer, Show Execution Plan)*.

A duplikált adatok problémája

Van egy elég gyakori feladat, amelynek megoldása elsô nekifutásból nem kézenfekvô. Nevezetesen, hogy keressük meg egy táblában az ismétlôdô sorokat. Leginkább ez is adatmigrációnál jön elô, amikor az SQL Serverbe bemásolt adathalmaz egy oszlopára szeretnénk ráadni egy UNIQE vagy PRIMARY KEY-t, de van benne egy-két ismétlôdô sor, ami megakadályozza ezt. Ilyenkor meg kell keresni azokat a sorokat, amelyekben azonosak a kérdéses oszlopok értékei. Ezzel ugye az a baj, hogy a WHERE egyszerre mindig csak egy sorral foglalkozik. Hogyan lehetne rávenni, hogy ugyanannak a táblának a sorait hasonlítgassa össze egymással? A megoldás egy self-join. „Sajnos" a Northwind egy konzisztens adatbázis, így abban nem fogunk találni olyan duplikált sorokat, amelyeket ki lehetne szûrni, mint logikailag hibásakat. Ezért a példa kedvéért nézzük a következô táblát *(Users)*:

```
nID FirstName LastName BirthDay
1   István    Király   1954-05-22
2   László    Lőrincz  1961-11-14
3   Jenő      Rejtő    1910-01-14
4   Jenő      Rejtő    1910-01-14
```

Tegyük fel, hogy ezt a táblát úgy kaptuk, hogy kiindulásként volt egy HR-tôl kapott konszolidálatlan felhasználói adatbázis Excel-ben, amit például a Data Transformation Services segítségével beimportáltunk egy Users nevûtáblába. Azért, hogy tudjunk hivatkozni a sorokra, adtunk mindegyiknek egyedi azonosítót egy szám formájában. Szeretnénk megtalálni azokat az embereket, akik többször is szerepelnek a táblában. Tegyük fel, hogy két sort *(és embert)* akkor mondunk azonosnak, ha a vezeték- és keresztnevük azonos, valamint egy napon születtek. Keressük hát meg, ki a kakukktojás!

```
SELECT
  u1.nID, u1.FirstName,
  u1.LastName, u1.BirthDay
FROM
  Users u1
INNER JOIN
  Users u2
ON
  u1.FirstName = u2.FirstName AND
  u1.LastName = u2.LastName AND
  u1.BirthDay = u2.BirthDay
WHERE
-- Máskülönben minden sornál megtalálná
-- önmagát, mint a sajt párját!
  u1.nID <> u2.nID
```

Az eredményen remélem senki nem lepôdik meg:

```
nID FirstName LastName BirthDay
3   Jenő      Rejtő    1910-01-14
4   Jenő      Rejtő    1910-01-14
```

Mi itt a szokatlan? Az, hogy illeszteni lehet több mezôre is, sôt nem csak numerikus mezôkre! Általában az él a legtöbb fejlesztôben, hogy biztos, ami biztos, minden táblához generálok egy mesterséges kulcsot *(Surrogate Primary Key)*, és azon keresztül illesztem a táblákat. Ez helyes, ha gyors adatbázist akarunk építeni, de azért nem csak egy egész szám lehet gyors. Mi van például a 2-3-4 betûs azonosítókkal? Azokat talán lassabb összehasonlítani, mint az egészeket? Nem sokkal, viszont sokkal könnyebben átlátható adatbázist kapunk.
Például egy felhasználói adatbázisban az osztály, ahol az alkalmazott dolgozik valószínûleg egy külön táblában lesz eltárolva. Numerikus kulcsokat használva az IT 1 lesz, a Finance 2 satöbbi. Ezzel szemben egy 3 karakteres azonosítóval az IT lehet 'IT', a Finance 'Fin', a Human Resources 'HR' és így tovább. Így a felhasználói táblát böngészve nem 'Kiss József', '2'-t fogunk látni, hanem 'Kiss József', 'Fin'-t, ami azért könnyebben dekódolható, nem?

Zárszó

A következô számban még további lekérdezéseket írunk a GROUP BY, a HAVING és társaik segítségével, hogy csodaszép statisztikákat tudjunk generálni. Mindenkit visszavárok!

Soczó Zsolt MCSE, MCSD
Protomix Rt.

Összetett lekérdezések

Az elôzô cikkünkben belemélyedtünk az illesztések lelkivilágába. Megnéztük, hogy egymásba ágyazott lekérdezésekkel milyen egyszerûen meg lehet oldani bonyolult problémákat is. Most további igen hasznos nyelvi elemekkel ismerkedünk meg, amelyek segítségével csoportosíthatjuk adatainkat, mûveleteket végezhetünk a csoportokkal, és nagyon látványos riportokat tudunk készíteni. Ehhez kapcsolódóan megnézzük, hogy a szerverbe beépített függvények segítségével mennyire át lehet alakítani az adatok jelentését.

Csoportosítsunk!

Bemelegítésül nézzük meg az alábbi lekérdezést:

```
SELECT
  od.OrderID, p.ProductName,
  od.UnitPrice * od.Quantity AS Amount
FROM
  [Order Details] od
INNER JOIN
  Products p
ON
  od.ProductID = p.ProductID
ORDER BY
  OrderID

OrderID  ProductName      Amount
10248    Queso Cabrales   168.00
10248    Singaporean       98.00
10248    Mozzarella       174.00
10249    Tofu              167.40
```

Egyszerûen kilistáztuk a megrendeléseket, kiszámolva az adott tétel értékét *(od.UnitPrice * od.Quantity)*. Szép ez a lista, csak túl részletes. Például a 10248-as megrendeléshez három sort listázott ki, mert a megrendelés három altételbôl állt. Egy riportban nem érdekesek az ilyen részletek, általában csak arra van szükség, hogy egy megrendelés összesen mekkora értékû volt. Elsô felindulásunkban a már ismertetett SUM függvényt használnánk, ami képes arra, hogy összegezze a tételeinket:

```
SUM(od.UnitPrice * od.Quantity) AS Amount
```

Persze ez nem azt tenné, amit várnánk tôle, hanem az összes megrendelés értékét összeadná, és eredményül egy számot kapnánk, amelyben minden megrendelés együttes értéke lenne. Mi lenne, ha lenne olyan utasításunk, amivel megmondhatnánk, hogy csoportosítsa a sorokat az OrderID mezô alapján, és a csoportokra végezze el az összegzést? Természetesen van ilyenünk, a GROUP BY az. Segítségével a GROUP BY mögé írt oszlopok szerint történik az eredményhalmaz csoportosítása, azaz az azonos OrderID-jú sorokból egyet készít, és a csoportokra kiszámítja az aggregált eredményeket.
Alakítsuk át a példánkat úgy, hogy megrendelésenként összegzett listát készítsen a GROUP BY és a SUM segítségével:

```
SELECT
  od.OrderID,
  SUM(od.UnitPrice * od.Quantity)
  AS Amount
FROM
  [Order Details] od
INNER JOIN
  Products p
ON
  od.ProductID = p.ProductID
GROUP BY
  od.OrderID
ORDER BY
  OrderID

OrderID     Amount
10248         440.00
10249        1863.40
10250        1813.00
```

Nagyszerûen mûködik! Miért vettem ki a p.ProductName oszlopot? Azért, mert semmi értelme, hisz pont az volt a célunk, hogy termékektôl és megrendelés tételektôl független listát kapjunk. Ha ezt elfelejtenénk, figyelmeztetni fog a szerver:

```
Column 'p.ProductName' is invalid in the se-
lect list because it is not contained in ei-
ther an aggregate function or the GROUP BY
clause.
```

Azaz a fordító a p.ProductName-et csak akkor fogadja el SELECT mögött, ha azt vagy felsoroljuk a GROUP BY-ban, vagy egy aggregáló függvénybe foglaljuk bele. Az elôbbinek az lenne a következménye, hogy a megrendelés tételeken belül termékenként tovább lenne bontva a részösszeg. Sokszor ez is cél lehet. A második javaslattal kapcsolatban: nehéz lenne olyan beépített aggregáló függvényt keresni, ami a terméknéven valami hasznosat tudna végezni. Úgyhogy ezt felejtsük el.
Mi van, ha csak azokat a megrendeléseket akarjuk kilistázni, amelyek összmegrendelés értéke nagyobb, mint 1000$? A WHERE használata sajnos nem vezet eredményre, mert az csak az egyes Order Details sorokban található értékekre tud szûrni, és nem pedig azok összegére. Másképpen fogalmazva valami ilyesmit szeretnénk látni a WHERE-ben:

```
WHERE
  SUM(od.UnitPrice * od.Quantity) > 1000
```

vagy

```
WHERE Amount > 1000
```

Csakhogy a WHERE-ben nem lehet használni aggregáló függvényeket, így a SUM-ot sem. Hogyan juthatunk túl ezen a dilemmán? Úgy, hogy van egy olyan speciális záradék *(clause)*, amelyet arra találtak ki, hogy a GROUP BY által definiált csoporton lehessen vele feltételeket érvényesíteni. Ez a záradék a HAVING. A HAVING nagyon hasonló a WHERE-hez, a különbség abban rejlik, hogy a záradékok után álló kifejezés mikor kerül kiértékelésre. A WHERE után álló kifejezést a csoportosítás elôtt értékeli ki a végrehajtó egység, azaz a WHERE segítségével elôre kiválogatjuk azokat a sorokat, amelyeket csoportosítani szeretnénk. Ezután jön maga a GROUP BY-ban elôírt csoportosító mûvelet. Létrejönnek a csoportok, valamint kiszámítódnak a csoportokra kijelölt aggregáló kifejezések. Ekkor jön a képbe a HAVING. A parancsvégrehajtó kidobálja azokat a csoportokat, amelyekre nem teljesül a HAVING után álló feltétel. A szenzációs az a dologban, hogy a HAVING után használhatunk aggregáló függvényeket, ellentétben a WHERE-el!
Mivel a WHERE segítségével drasztikusan le lehet csökkenteni a csoportosítandó sorok számát, ezért érdemes minden olyan feltételt, ami nem a csoportokra vonatkozik a WHERE-be rakni a HAVING helyett. Ha ellenkezôen cselekszünk, a fordító nem fog figyelmeztetni minket. ô szolgai módon végrehajtja a lekérdezést az általunk elôírt módon. A felhasználók viszont szólnak majd az alkalmazásunk lassúsága miatt...
Szerencsére azonban a Query Optimizer ennél okosabb. Általában, hangsúlyozom, általában észreveszi, hogy nem optimálisan írtuk meg a lekérdezést, és a nem megfelelô helyre írt kifejezéseket a végrehajtás idejére átrakja a megfelelô helyre. Még sokszor tapasztaluk majd, ahogy az SQL Server fejlesztôk intelligenciája igyekszik pótolni a buta alkalmazásfejlesztôét. Hogy érthetôbb legyen a HAVING és a WHERE közötti különbség, egy táblázatban összefoglaltam, hogy milyen helyzetben melyik záradékot használhatjuk.

	WHERE	HAVING
Mikor szûr?	A csoportosítás elôtt	A csoportosítás után
Mit szûr?	Sorokat	Csoportokat
Tartalmazhat-e aggregáló függvényeket?	Nem	Igen

Nézzünk egy példát, amelynek segítségével közelebb kerülhetünk a GROUP BY és a HAVING szelleméhez. Lássunk egy elég bonyolult lekérdezést, amiben minden benne van, amit eddig tanultunk:

```
SELECT
  p.ProductID, p.ProductName,
  'Amount' = SUM(od.UnitPrice *
   od.Quantity)
FROM
  [Order Details] od
INNER JOIN
  Orders o
ON
  o.OrderID = od.OrderID
INNER JOIN
  Products p
ON
  p.ProductID = od.ProductID
WHERE
  o.OrderDate >= '1998.05.05' AND
  o.OrderDate <= '1998.05.07'
GROUP BY
  p.ProductID, ProductName
HAVING
  SUM(od.UnitPrice * od.Quantity) > 800
ORDER BY
  Amount DESC
```

Az SQL kód magyarra fordítása: készítsünk egy olyan listát, amely az 1998. május ötödike és hetedike közötti megrendelések összértékét listázza ki, termékenkénti bontásban. Csak azokra a termékekre vagyunk kíváncsiak, amelyek megrendeléseinek összege nagyobb, mint 800 dollár. A lista legyen rendezve az eladási érték alapján, csökkenô sorrendben. Huh, magyarul nehezebb megfogalmazni, mint SQL-ül! Nézzük meg a kimenetét:

```
ProductID ProductName     Amount
64        Wimmers gute    4389
 2        Chang           1178
16        Pavlova          802
```

Néhány szó a szintaktikával kapcsolatban. Az

```
'Amount' = SUM(od.UnitPrice *
   od.Quantity)
```

kifejezés a

```
SUM(od.UnitPrice *
   od.Quantity) AS Amount
```

kifejezéssel egyenértékû. Lehet így is írni, meg úgy is írni. Használjuk az, amelyik olvashatóbb számunkra. A leglustábbak a második formát használják, úgy, hogy még az AS-t is elhagyják *(én is ilyen vagyok)*.
Az ORDER BY-ban az Amount álnevet használhattuk a bonyolult SUM(od.UnitPrice * od.Quantity) helyett. Ezt a szintaktikai könnyítést sajnos csak az ORDER BY-ban használhatjuk ki, a HAVING-ben már nem. Kár.
Az

```
o.OrderDate >= '1998.05.05' AND
o.OrderDate <= '1998.05.07'
```

szûrôfeltételt elegánsabban is megfogalmazhatjuk a BETWEEN operátor segítségével:

```
OrderDate BETWEEN
 '1998.05.05' AND '1998.05.07'
```

A két kifejezés logikai értéke azonos.
Gyakori kérés a marketing vagy a pénzügy részérôl, hogy olyan összesített statisztikát kérnek, amelyben az eladási adatok napi bontásban láthatók. Mivel a generálódó listát emberek fogják kiértékelni, ezért ôket elsôsorban az irányvonalak érdeklik, nem pedig az összes részeredmény az utolsó bitig. Így például feltételként szabják, hogy a napi listában csak azok a termékek szerepeljenek, amelyekbôl több mint

nehézségi fok:

egyet rendeltek meg egy adott napon *(a nap slágere)*:

```
SELECT
  OrderDate,
  p.ProductName,
  'Amount' =
  SUM(od.UnitPrice * od.Quantity),
  COUNT(*) AS OrderedProducts
FROM
... --ugyanaz, mint az előző lekérdezésben
WHERE
  OrderDate BETWEEN
  '1998.05.01' AND '1998.05.07'
GROUP BY
  OrderDate, p.ProductID, ProductName
HAVING
  COUNT(*) > 1
ORDER BY
  OrderDate, Amount DESC

OrderDate  ProductName Amount OProd
1998-05-05 Chang        532    2
1998-05-06 Chang        646    2
1998-05-06 Grandma's    525    2
1998-05-06 Tofu         488    2
```

Ebbôl olyan okosságokra lehet következtetni, hogy a Chang nagyon finom lehet, mert ötödikén és hatodikán is megrendeltek belôle kettôt is! Ennél tovább azonban nem megyünk, ez nem a mi szakmánk.
Szakmai szempontból az utolsó oszlop érdekes a számunkra: **`COUNT(*) AS OrderedProducts.`** A **`COUNT(*)`** a többi aggregáló függvényhez hasonlóan másként viselkedik, ha GROUP BY van a közelben: nem az egyedi sorokat számolja meg, hanem a csoportokon belüli sorok számát. Másképpen: nem a teljes eredményhalmazra ad egy eredményt, hanem minden egyes csoportra külön-külön. Pont ez az, ami nekünk kellett, és a **`HAVING`** volt olyan szíves aggregáló függvényt beengedni a feltételek közé. Hurrá!

Beépített skaláris függvények

Mi is az a skaláris függvény? A függvény állatfaj azon alfaja, amely egy értékbôl egy értékre képez le. Azaz nem olyan, mint a SUM volt, ami sok sor tartalmát összegezve adott vissza egyetlen számot, mert ô az aggregáló típusú függvények képviselôje, azaz, amelyek több értékbôl állítanak elô egyet. Inkább gondoljunk az abszolút értéket képzô ABS függvényre *(a blokkolásgátló függvény :)*. **`ABS(-4) = 4`**. Azaz a mínusz négyet leképezte plusz négyre. Nagy csodák vannak ebben a Serverben!
Az SQL Server nagyon sok beépített, skaláris függvénnyel rendelkezik. Vannak matematikai célúak: abszolút érték *(ABS)*, trigonometrikus függvények *(SIN, COS, TAN, COT, valamint ezek arcus megfelelôi)*, logaritmus függvények *(LOG, LOG10)*, exponenciális függvény *(EXP)*, kerekítések *(ROUND, FLOOR, CEILING)*, véletlen szám generáló függvény *(RAND)* stb. Majdnem olyan gazdag a kínálat, mint más „polgári" nyelvekben, mint pl. a Visual Basic-ben.
Van nagyon sok szövegkezelô függvény: különbözô darabolások *(LEFT, RIGHT, SUBSTRING)*, szövegdarabok keresése *(PATINDEX, CHARINDEX)*, kis-nagybetû konvertálók *(UPPER, LOWER)*, számformátumról szövegre átalakító *(STR)*, kezdô és záró szóközt levágó *(LTRIM, RTRIM)*. Vannak egzotikusabbak is: REVERSE, ami megfordít egy szöveget: „cirmos" ‡ „somric". Van olyan, ami nagyon hasznos lenne, ha magyarul is mûködne, csak hát a magyar nyelv elég ellenálló a formalizálással szemben: a DIFFERENCE és a SOUNDEX. A DIFFERENCE egy 0-4-es skálán képes megmondani két szövegrôl, hogy kimondva, hangzásban *(!)* mennyire hasonlítanak. Nem a karakterláncok írt, hanem kimondott formája. Ez a szolgáltatás nagyon jól jönne például egy telefonkönyv alkalmazásnál, ahol nem lehet tudni, hogy pontosan hogyan írtak egy nevet, de például valahogy úgy hangzott, hogy „socó". A „Smith" és a „Smythe"-re a például DIFFERENCE azt mondja, hogy a távolságuk 4, azaz nagyon hasonlítanak. Az „other" és a „brother" szavak 2 távolságra vannak egymásra, azaz még hasonlítanak, de azért nem annyira. Nagyon jó lenne ez magyarul is! Így talán a Gizike és a gôzeke is kaphatna egy 1-est.
Tovább a függvények útján. Nagyon hasznosak, bár ritkábban használatosak az úgynevezett metaadat-függvények. Ezek a rendszertáblákból kérdeznek le adatokat *(manuálisan tilos, mert bármelyik szervizcsomag megváltoztathatja)*, melynek segítségével belsô információkat lehet megtudni az adatbázisunk tulajdonságairól. Ha például az alkalmazásunk kíváncsi, hogy a Employee nevû tábla FirstName nevû oszlopa hány byte-ot foglalhat el az adatbázisban:

```
COL_LENGTH ('Employee', 'FirstName')
```

Ez egy nvarchar*(50)*-es oszlopra 100-at adna eredményül *(az nvarchar Unicode formátumú, azaz minden karakter 2 bájtot foglal el)*.
További függvénykategóriákat is találunk még a Serverben, de ezeket terjedelmi okok miatt most nem közöljük. A Books Online-ban részletesen dokumentálva vannak mindannyian.
Végül, de nem utolsó sorban beszéljünk az egyik leghasznosabb függvénycsaládról: a dátumkezelô függvényekrôl. Ezek megérnek a többinél kicsit több figyelmet.

Dátumzsonglôrködés

Eddig elég egyszerû volt bevetni a GROUP BY-t, mivel midig volt egy olyan oszlop, amire természetesen lehetett csoportosítani. Azonban ilyen nem mindig létezik. Például az elôzô példában az OrderDate napra kerekített érték volt, így könnyû volt napra csoportosítani, mivel csak be kellett írni a GROUP BY-ba. De mit teszünk, ha heti bontásban várják a kimenetet? Ha nem ismerjük a DATEPART függvényt, akkor bajban leszünk. De ha igen, akkor:

```
SELECT
  DATEPART(wk, OrderDate) AS WeekNum,
  'Amount' = SUM(od.UnitPrice *
  od.Quantity),
  COUNT(*) AS OrderedProducts
FROM
...
WHERE
  OrderDate BETWEEN
  '1998.01.01' AND '1998.02.01'
GROUP BY
  DATEPART(wk, OrderDate)
HAVING
  COUNT(*) > 1
ORDER BY
  WeekNum, Amount DESC

WeekNum     Amount          OrderedProducts
1            4691.0000      12
2            30894.6600     30
3            18822.6700     38
4            24330.5400     38
5            22115.8500     34
```

A DATEPART függvény az egyik leggyakrabban használt beépített függvény. Visszatér egy egész számmal, ami a date paraméterben megadott dátum egy bizonyos darabjának felel meg. A használata nagyon egyszerû:

```
DATEPART(datepart, date)
```

ahol a datepart a következô kifejezések valamelyike lehet:

Jelentés	Teljes név	Rövidítés
Év	year	yy, yyyy
Negyedév	quarter	qq, q
Hónap	month	mm, m
Az év n. napja	dayofyear	dy, y
Nap	day	dd, d
Hét	week	wk, ww
A hét n. napja	weekday	dw
Óra	hour	hh
Perc	minute	mi, n
Másodperc	second	ss, s
Ezredmásodperc	millisecond	ms

A függvényben használhatjuk mind a teljes nevet, mind a rövidítést.
A példánk bizony sánta. Mivel minden évben van 1. hét, 2. hét, satöbbi, ezért a lekérdezésünk össze fogja vonni az összes év ugyanazon hetébe esô eladásokat. Ez természetesen nem helyes, és ezt a problémát úgy fogjuk orvosolni az utolsó, szinte tökéletes lekérdezésünkben, hogy a hét sorszáma mellé felsoroljuk az évet is mind a SELECT utáni listában, mind a GROUP BY-ban, így egyértelmûen azonosítva lesz a hét.
Az elôzô példánkban arra voltunk kíváncsiak, hogy az adott dátum az év hányadik hetébe esik. Azonban az SQL Servert Amerikában írták, ahol a hét elsô napja a vasárnap, nem pedig a hétfô. ôk tudják, mi a jó nekik, azonban a DATEPART is ennek megfelelôen mûködik, aminek mi nem örülünk. Mivel azonban a Microsoft nem csak Amerikában akarja eladni az SQL Servert, ezért beépítette annak lehetôségét, hogy megváltoztassuk a hét elsô napját:

```
SET DATEFIRST 1
```

Ennek hatására a hét elsô napja ismét a hétfô lesz, így az összes dátumkezelô függvény helyesen fog mûködni. Az SQL Serverben sok, a fentihez hasonló beállítás létezik, amelyekkel a szerver alapértelmezett viselkedését változtathatjuk meg. Ezekkel egy késôbbi cikkben még részletesen foglalkozunk.
A dátumkezelô függvények további igen hasznos képviselôje a DATEADD függvény. Ez, mint a neve is sugallja, arra való, hogy egy dátumhoz hozzáad valamilyen idôintervallumot. Mi határoz meg egy idôintervallumot? A hossza és a mértékegysége. Ennek megfelelôen a függvény formátuma a következô:
DATEADD *(datepart , number, date)*
A datepart az elôzô táblázatban közölt értéket veheti fel, a weekday kivételével, mert annak nincs semmi értelme ebben az összefüggésben. Szerintem a dayofyear-nek sincs, de az úgy mûködik, mintha day-t írtunk volna. A number egy egész szám, ami az intervallumot írja le. A date pedig a kiinduló dátum. Nézzük meg mûködés közben:

```
SELECT DATEADD(day, 1, '2000/01/05 18:12')
2000-01-06 18:12:00.000
SELECT DATEADD(mi, 5, '2000/01/05 18:12')
2000-01-05 18:17:00.000
SELECT DATEADD(hh, -3, '2000/01/05 18:12')
2000-01-05 15:12:00.000
DECLARE @d DATETIME
SET @d = '2000/01/05 18:12'
SELECT @d + 3
2000-01-08 18:12:00.000
```

Az elsô három példa morális tanulsága: nincs DATESUB, a DATEADD-ot kell negatív számmal meghívni.
Az utolsó példa ravasz. Egy dátum típusú mezôhöz hozzáadunk 3-at, egy egész számot, aminek az a jelentése, hogy a dátumot megnöveli 3 nappal. Érdekes, de ha valaki nem tudja explicit a + operátor e polimorf tulajdonságát, az meglepôdhet a kódunkon.
A harmadik hasznos dátumkezelô függvény a DATEDIFF. Formátuma:

```
DATEDIFF (datepart, startdate, enddate)
```

Azaz a startdate és enddate dátumok különbségét adja vissza a datepart-ben definiált egységben:

```
--Hány másodperc is egy nap?
SELECT    DATEDIFF(second,    '2000.01.01',
'2000.01.02')
86400
```

Dátum agytorna

Szép lett az elôzô példánk listája, de nekem például nem sokat mond az, hogy most a 38. hétben járunk. A menzán és a hivatalokban gyakran láthatjuk ezt a fajta idômeghatározást, de nekem sokkal szimpatikusabb lenne a lista, ha a hetet jelzô szám mellett ott láthatnám azt is, hogy mely dátumhatárok zárják az adott hetet. Próbáljuk meg kitalálni, hogyan le-

Microsoft BackOffice

hetne ezt összerakni Transact SQL-ben. Nem lesz triviális, kéretik egy dupla KV-t inni a következôk elôtt!
Adott a dátumunk, tároljuk ezt a @d változóban. Azt, hogy ez a dátum az év hányadik hetébe esik, a DATEPART(week, @d) függvénnyel könnyedén megtudhatjuk. Hogyan lesz ebbôl meg a keresett hét kezdô dátuma? Úgy, hogy valahogyan meg kellene találni az adott év elsô hétfôjét, és ahhoz hozzá kellene adni annyiszor 7 napot, ahányadik héten járunk az elsô hétfôhöz képest. Az év eleji tört hét is hétnek számít! Nézzük meg mindezt Transact SQL-ben!

```
--A hét első napja a hétfő legyen
SET DATEFIRST 1
--Ehhez a dátumhoz keressük a hetet és a
--hetet záró határokat
DECLARE @d DATETIME
--Ebben lesz az év első napjának dátuma
--(nem első hétfő, hanem január elseje!)
DECLARE @dFirstDayOfYear DATETIME
--Teszt dátum. Ez egy keddi nap a 2. héten
SET @d = '2000/01/04'
--A dátumhoz tartozó hét tesztelése
SELECT DATEPART(week, @d)
2
--Az év első napjának megkeresése
SET @dFirstMondayOfYear = CONVERT(CHAR(4), @d,
112)
SELECT @dFirstDayOfYear
2000-01-01 00:00:00.000
```

Itt álljunk meg egy pillanatra. Mi az a CONVERT függvény, és mit jelent a 112-es paraméter? A CONVERT a különbözô adattípusok közötti konverzióra való. Különösen akkor hasznos, ha dátum formátumot kell szöveggé konvertálni. Az elsô paramétere mondja meg, hogy milyen típussá szeretnénk konvertálni. Itt char(4)-et adtunk meg, ami 4 karaktert képes tárolni. A második paraméter a konvertálandó kifejezés, a harmadik pedig a konvertált eredmény formátumát szabályozza. Dátum bemenet és szöveg kimenet esetén a 112 azt jelenti, hogy a dátumot yyyymmdd formátumra konvertálja át. De hisz az eredmény 8 karakter, mi meg char(4)-et adtunk meg! Ez benne a trükk. 2000. 01. 04.-ból 20000104 lenne, de mivel a char(4) csak az elsô 4 karaktert képes eltárolni, a maradék négy egyszerûen elveszik. Azaz mi lesz a konverzió eredménye? "2000". De akkor miért kaptunk a SELECT @dFirstDayOfYear eredményeként 2000-01-01-et? Azért, mert a @dFirstDayOfYear dátum típusú, és a szerver a „2000" sztringet 2000. január 1-é konvertálta. Implicit módon, azaz anélkül, hogy erre külön megkértük volna. Ez azért egy kicsit piszkos munka volt. Inkább segítsünk neki:

```
SET @dFirstDayOfYear = CONVERT(CHAR(4), @d,
112) + '.01.01'
2000-01-01 00:00:00.000
```

Na, ez így már szép. De menjünk tovább. Hogyan kapjuk meg ebbôl az elsô hétfôt? Ha tudjuk, hogy január elseje a hét hányadik napja, akkor ebbôl már könnyû kiszámolni, hogy az elsô hétfô hányadikára esik: menjünk vissza az év elsô napjától annyi napot, ahányadik napra esik az a hétben, és adjunk hozzá 8-at. Például 2000. január elseje szombat volt, ami a hét 6. napja.

```
SELECT DATEPART(weekday, @dFirstDayOfYear)
6
```

Ha visszamegyünk 6 napot, az 1999. december 26-a, ami a 2000. év elsô hétfôjét megelôzô hétfô elôtti nap *(vasárnap)*.

```
SELECT    DATEADD(day,    -DATEPART(weekday,
@dFirstDayOfYear), @dFirstDayOfYear)
1999-12-26 00:00:00.000
```

Ehhez már csak hozzá kell adni 8 napot, és meglesz a 2000. év elsô hétfôje.

```
SELECT    DATEADD(day,    -DATEPART(weekday,
@dFirstDayOfYear)+8, @dFirstDayOfYear)
2000-01-03 00:00:00.000
```

Eljutottunk a tárgy év elsô hétfôjéig. Most már nincs más dolgunk, mint ehhez hozzáadni annyiszor 7 napot, ahányadik héthez keressük a hetet kezdô hétfôt.

```
SELECT    DATEADD(day,    (-DATEPART(weekday,
@dFirstDayOfYear)+8)  +  (DATEPART(week,  @d)-
2)*7, @dFirstDayOfYear)
2000-01-03 00:00:00.000
```

Azért kellett kettôt kivonni a hét számából, mert 1-tôl kezdôdik a hetek számozása és nem 0-ától, valamint, mert az aktuális napot megelôzô hétfôre vagyunk kíváncsiak, nem pedig a következôre. A záró napot innentôl kezdve gyerekjáték meghatározni, csak nem kettôt, hanem egyet kell kivonni a hetek számából.

```
SELECT    DATEADD(day,    (-DATEPART(weekday,
@dFirstDayOfYear)+8)  +  (DATEPART(week,  @d)-
1)*7, @dFirstDayOfYear)
2000-01-10 00:00:00.000
```

Alakítsuk át a korábbi GROUP BY-os példánkat úgy, hogy a hetek száma mellé legyen kiírva azok kezdete és vége is. Ehhez az elôbb kiagyalt kifejezéseket össze kell vonni, és be kell írni a megfelelô helyre a kiinduló lekérdezésben, valamint rakjuk bele az évet is, ahogy korábban ígértük:

```
SET DATEFIRST 1
SELECT
  DATEPART(year, OrderDate) AS YearNum,
  DATEPART(wk, OrderDate) AS WeekNum,
  DATEADD(day, (-DATEPART(weekday,
  CONVERT(CHAR(4), OrderDate, 112) +
  '.01.01')+8) + (DATEPART(week,
  OrderDate)-2)*7, CONVERT(CHAR(4),
  OrderDate, 112) + '.01.01') AS StartDay,
  DATEADD(day, (-DATEPART(weekday,
  CONVERT(CHAR(4), OrderDate, 112) +
  '.01.01')+8) + (DATEPART(week,
  OrderDate)-1)*7, CONVERT(CHAR(4),
  OrderDate, 112) + '.01.01') AS EndDay,
  'Amount' = SUM(od.UnitPrice *
  od.Quantity),
  COUNT(*) AS OrderedProducts
FROM
...
WHERE
  OrderDate BETWEEN
  '1997.12.22' AND '1998.01.18'
GROUP BY
  DATEPART(year, OrderDate),
  DATEPART(wk, OrderDate),
  DATEADD(day, (-DATEPART(weekday,
  CONVERT(CHAR(4), OrderDate, 112) +
  '.01.01')+8) + (DATEPART(week,
  OrderDate)-2)*7, CONVERT(CHAR(4),
  OrderDate, 112) + '.01.01'),
  DATEADD(day, (-DATEPART(weekday,
  CONVERT(CHAR(4), OrderDate, 112) +
  '.01.01')+8) + (DATEPART(week,
  OrderDate)-1)*7, CONVERT(CHAR(4),
  OrderDate, 112) + '.01.01')
ORDER BY
  WeekNum, Amount DESC

YN    WN StartDay    EndDay      Amount OP
1997  52 1997-12-22 1997-12-29 17678   32
1997  53 1997-12-29 1998-01-05 14871   18!
1998  1  1997-12-29 1998-01-05  4691   12!
1998  2  1998-01-05 1998-01-12 30894   30
1998  3  1998-01-12 1998-01-19 18822   38
```

Konklúzió

Látható, hogy az évváltásnál nem jól mûködik a dátumkezelô algoritmusunk. Ezt még tökéletesíteni fogjuk a következô számban. És miért lett ennek az egyszerû feladatnak a megoldása ilyen bonyolult, annak ellenére, hogy nem is tökéletes? Azért, mert majdnem ugyanazt a hosszú kódrészletet négyszer egymás után le kellett írnunk, szinte változatlanul. De hát nem azt tanultuk az iskolában, hogy az ismétlôdô kódrészeket függvényekbe kell rakni? De! És nem arról regéltek nekünk, hogy a függvények paraméterezésével még a nem teljesen azonos kódrészeket is össze lehet vonni? De, de, de! Akkor miért nem élünk ezzel a lehetôséggel? Microsoft SQL Server 7-ig azért nem, mert nem volt meg a módunk erre, mert nem voltak User Defined Function-ök, felhasználói függvények. De SQL 2000-ben végre vannak, és pont az ilyen problémákra adnak igen elegáns megoldást. De errôl majd a következô részben.

Soczó Zsolt MCSE, MCSD, MCDBA
Protomix Rt.

Transact SQL *(IV. rész)*

Bevezetés

Még be sem fejezôdött az SQL Server 7 fejlesztése, és máris több oldalas volt az SQL Server fejlesztô csapat „kívánságlistája“, azaz, hogy a megoldásszállító fejlesztôk milyen funkciókat szeretnének látni a következô SQL Server verzióban. Ennek eredményeként született – az XML támogatás mellett – az SQL 2000 legnagyobb újítása a felhasználói függvények formájában. A cikkben nagyon tömören megnézzük a téma elméleti hátterét, hogy azután megírjunk néhány függvényt, amelyek segítségével szövegeket manipulálunk, megírjuk a Basic Split függvény SQL párját, megtanulunk körlevelet küldeni felhasználói függvényekkel, és kifejlesztünk egy behatolásjelzô programot. Hosszú, de nagyon izgalmas rész lesz ez cikksorozatunkban, érdemes végigolvasni!

Minden, amit tudni akartál a felhasználói függvényekrôl, de nem merted megkérdezni

Ez a rész a felhasználói függvények lelkivilágával, formai és mûködésbeli tulajdonságaival foglalkozik. Aki tudja, miért fontos a determinizmus kérdése a függvényeknél, az nyugodtan ugorjon az utolsó fejezetre, ahol fokozatosan bonyolódó példákat találhat. Aki nem, az tartson velem a következô részekben is.

Az SQL Serverben nagyon sok beépített függvény található *(lásd cikksorozatunk elôzô része)*, azonban ezek nyilvánvalóan nagyon általános függvények, mint például a *(ruhaiparból ismert)* LEN függvény, ami egy szöveg hosszát adja vissza. Nagyon jók ezek a beépített függvények, köszönjük ôket, de a gyakorlati problémák megoldásához – pont az általános voltuk miatt – nem elegek. Mint építôkockák kitûnôek, de hogyan építünk belôlük várat? Nos, hosszú várakozás után a Microsoft elékészítette a habarcsot, megalkotta a felhasználói függvényeket, így most már semmi akadálya, hogy megalkossuk a saját PAMUT vagy a GYAPJU nevû függvényeinket, amelyek belsô mûködését mi írhatjuk elô.

Egyszerûen megfogalmazva a felhasználói függvény olyan Transact SQL utasítások sorozata, amelyeket azért csomagolunk egybe, hogy több helyen is felhasználhassuk. Nagyon jól kiegészítik a tárolt eljárásokat, mert minden olyan helyen felhasználhatjuk ôket, ahol a beépített függvényeket is, azaz ahol a tárolt eljárásokat legtöbbször nem. A legegyszerûbb példa erre a SELECT-ben való felhasználás. Például, ha van egy osszeadas nevû függvényünk, akkor azt felhasználhatjuk két oszlopban található számok összeadására, a SELECT utasítás részeként:

```
SELECT osszeadas(Ár, ÁFA), Termék FROM ...
```

Ennél kevésbé kézenfekvô helyeken is használhatjuk a függvényeinket: WHERE feltételben, HAVING-ben, CHECK CONSTRAINT-ekben, DEFAULT CONSTRAINT-ekben, számított oszlopok képzésében. Mindenhol mûködnek, ahol a szerver valamilyen kifejezést vár *(mint a>b, c=4 vagy 2x2=5)*.

Azok kedvéért, akik nem szeretnek tömény oldalakat kódok nélkül látni, megmutatom az elôbbi függvény deklarációját. Részletes magyarázatot a cikk második felében talál a lelkes Olvasó.

```
CREATE FUNCTION osszeadas
(
  @a INT,
  @b INT
)
  RETURNS INT
BEGIN
  RETURN @a + @b
END
```

Az SQL Server a függvényeket sokszor tranzakciók, illetve SELECT, UPDATE, satöbbi utasítások kellôs közepén hívja meg. Emiatt rendetlen az a függvény, ami menet közben módosítja egy tábla tartalmát, miközben egy SELECT *(ami ôt hívta meg)* éppen dolgozik rajta – nos ilyen esetben nagy lárma és kalamajka támadhatna. Az SQL Server azonban nem keresi a bajt, ezért megpróbálja megkötni a kezüket, hogy ne csináljunk felfordulást. Azaz a felhasználói függvényekben nem tehetünk meg akármit, csak a következôket:

- Definiálhatunk saját változókat és kurzorokat a DECLARE utasítással. Csak lokális kurzorokat készíthetünk így, globálisakat, azaz amelyek a függvény lefutása után is léteznének nem.
- A függvényben deklarált lokális változóknak értéket adhatunk *(naná, e nélkül akár ki is dobhatnák a függvényeinket)*.
- Használhatunk kurzormûveleteket, de csak úgy, hogy a FETCH utasítás eredményeit lokális változókba rakjuk el *(a kurzorokkal egy teljes cikk fog foglalkozni a következô hónapban)*.
- Bevethetjük a programfolyam-vezérlô utasításokat: if, then, for, while, goto, satöbbi. Ezek nélkül nem is lehetne egy komolyabb függvényt megírni.
- Alkalmazhatjuk az összes adatmódosító utasítást *(INSERT, UPDATE, DELETE)*, ha azok csak lokális táblákon végeznek mûveleteket. Ebbôl következôen nem lehet módosítani külsô táblákat. Természetesen lekérdezésekben szerepelhetnek.
- Meghívhatunk külsô tárolt eljárásokat *(Extended Stored Procedure)* az EXECUTE utasítással. „Hagyományos“ tárolt eljárásokat nem lehet meghívni belôlük, hisz azokból már könnyedén beavatkozhatnánk a „külvilágba“.

Látható, hogy minden pontban arról van szó, hogy megtehetünk szinte bármit, amit csak akarunk, de csak lokálisan, azaz a függvény nem avatkozhat be a külvilágba. Van egy kis szemétdombunk, ott kapirgáljunk. Bár az utolsó pont, azaz, hogy külsô tárolt eljárásokat is meghívhatunk, azért egy nagyon tág fogalom. Mert mit csinálhat egy külsô tá-

rolt eljárás? Bármit! Amit akar. Azaz például megteheti azt, hogy visszafelé nyit egy kapcsolatot a kiszolgálóra, és azon keresztül megváltoztatja azt a táblát, amiben éppen dolgozik a kódunk a függvény hívása során. De ez általában már túlmutat a normális használaton. Megtehették volna a fejlesztôk, hogy teljesen letiltják a külsô eljáráshívásokat, de akkor meg elestünk volna olyan nagyszerû lehetôségektôl, mint külsô parancsok meghívása *(xp_cmdshell)*, levélküldés *(xp_sendmail)* vagy event log írás *(xp_logevent) (és még sok egyéb hasznos funkció)*. Az imént felsorolt három külsô tárolt eljárás azonban pont olyan, aminek nem szabadna lefutni egy függvényben. Miért? Azért mert egy függvény nem változtathatja meg globálisan a rendszer állapotát. A rendszeren nem csak az SQL Server belsô lelkivilágát értjük, hanem az egész világot. Így például az xp_cmdshell segítségével akár le is formázhatjuk kollégánk merevlemezét. Fogadjunk, hogy megváltozik a kolléga *(lelki)* állapota. :) Azaz ezeket a külsô tárolt eljárásokat nem szabadna meghívni egy felhasználói függvénybôl, amire nyomatékosan fel is hívja a figyelmet a dokumentáció *(Books Online)*. Azonban, a fordító egy szót sem szól, ha olyan függvényt írunk, amiben felhasználjuk a veszélyes tárolt eljárások valamelyikét! Ezt még ki fogjuk használni a cikk végén található programokban. Pont olyan ez, mint a C programozás: ha meggondoltan csináljuk, miénk a világ. Ha nem, akkor csak General Protection Fault-okat generálunk.

A nemdeterminisztikus jövô

Vannak még más problémás elemek is, amelyeket bizonyos esetekben szintén nem szabad használni függvényekben. Ezek a nemdeterminisztikus függvények. Mik is ezek? Ôk a függvények azon fajtái, amelyeknek a mûködése vagy az általa visszaadott érték idôben vagy a szerver állapotától függôen nem megjósolható módon változik. Azaz ugyanazokkal a paraméterekkel meghívva egyszer a-t mond, másszor b-t. A legegyszerûbb példa erre a GetDate() beépített függvény, ami a pillanatnyi idôt adja vissza *(a GetTime szerencsésebb név lett volna)*. Ez minden egyes meghívás pillanatában más értéket ad vissza, legalábbis addig, amíg jár a gépünkben a kvarckristály. A fordítóprogram nem engedi meg, hogy ilyen nemdeterminisztikus beépített függvényeket helyezzük el a saját függvényeinkben. Például a következô függvény törzsre: RETURN RAND*(10)* a fordító az „Invalid use of 'rand' within a function." hibaüzenettel válaszol.

Miért ilyen problémás pont a determinizmus kérdése az SQL Serverben? Azért, mert vannak benne olyan új szolgáltatások, amelyek nem tudnának helyesen mûködni a „bizonytalan" nemdeterminisztikus függvényekkel. Két helyen nem lehet felhasználni a nemdeterminisztikus függvényeket:

- Indexelt számított oszlopokon, azaz, ha olyan oszlop ra szeretnénk indexet készíteni, amelynek érékei egy másik *(egy vagy több)* oszlopból származnak, és a számított érték valamilyen nemdeterminisztikus függvényen alapul.
- Olyan nézetekben, ahol a nézetre clustered indexet szeretnénk használni.

A két megszorítás alapján már eléggé érthetô, hogy miért kell foglalkozni a determinizmus kérdésével. Mindkét esetben indexet építünk táblában található adatokra. Próbált már valaki megülni egy vásári bikát? Nem egyszerû. Hasonló módon az SQL Server sem tud indextáblát építeni olyan adatokra, amelyek minden pillanatban változnak. A clustered index az adatok fizikai sorrendjét határozza meg. Ezen a héten így legyek sorban az adatok, a következô héten meg másképp, csak azért, mert meggondolta magát a transzformáló függvény? Na nem, ez nonszensz lenne. Ezért nem is tehetünk ilyet.

Ragaszkodás a barátokhoz

Egyetlen apró fogalom maradt már csak hátra, hogy ténylegesen megírhassuk elsô függvényünket. Ez a séma-kötés fogalma. A felhasználói függvények igen erôsen kötôdnek azokhoz a táblákhoz, és egyéb objektumokhoz, amelyekre hivatkoznak. Ha azok módosulnak anélkül, hogy errôl a függvény tudna, akkor a kapcsolatuk vége barátságtalan lesz, és a függvény nem fog jól mûködni. Azért, hogy a jó viszonyban ne következhessen be szakadás, a függvény létrehozásakor *(CREATE FUNCTION)* megadhatjuk, hogy a függvény legyen hozzákötve az általa használt objektumokhoz. Ezt az SQL Server megjegyzi, és nem engedi módosítani vagy törölni az ily módon leláncolt objektumokat. A kötés jelzését a RETURNS és a függvény törzsét kezdô BEGIN közé kell írni:

```
...
RETURNS ...
WITH SCHEMABINDING
BEGIN
...
```

Függvénytípusok

Háromféle felhasználói függvénytípust hozhatunk létre az SQL 2000-ben:

- Skaláris függvények, melyeknek visszatérési értéke skaláris, azaz egy érték *(scalar functions)*
- Egy utasításból álló, tábla visszatérési értékû függvények *(inline table valued functions)*
- Több utasításból álló, tábla visszatérési értékû függvények *(multi statement table valued functions)*

Az utóbbi két fajta nagyon hasonlít egymásra, mint ez a részletes tárgyalásból hamarosan kiderül.

Skaláris függvények

A skaláris függvények nagyon egyszerûek: kapnak néhány paramétert, azokon végeznek valamilyen mûvelet, majd az eredményt egy skaláris értékként visszaadják. Azaz visszaadnak egy számot, egy szöveget, egy dátumot satöbbi. Leginkább a procedurális nyelvek függvényeihez hasonlítanak.

Rutinos tárolt eljárás programozók! A felhasználói függvényeknek nincsenek kimeneti paramétereik! Azaz nem lehet valamelyik paramétert megjelölni, hogy az visszafelé fog majd valamilyen információt szolgáltatni a hívónak. Ezt a lehetôséget azért kellett bevezetni a tárolt eljárásoknál, mert azok csak egy egész számot tudnak visszaadni visszatérési értékként, így nem tudtunk volna például egy dátumot visszaadni a hívónak. Erre szolgáltak a kimeneti paraméterek. Hogy teljesen érthetô legyen, álljon itt egy tárolt eljárás, amelynek a harmadik paramétere kimeneti paraméter:

```
CREATE PROCEDURE osszead
  @a INT,
  @b INT,
  @c INT OUTPUT
AS
SET @c = @a + @b
--Eddig a tárolt eljárás deklarációja.
--Látható, hogy egy tárolt eljárásban nem
--kötelező a visszatérési értéket megadni
--Azaz lehetne egy záró RETURN ..., de
--nem szükséges, mert most nem használjuk
--fel a visszatérési értéket.

DECLARE @osszeg INT
--hívjuk meg

EXECUTE osszead 1,4, @osszeg OUTPUT
SELECT @osszeg
5  --Működik!
```

Nos, kimeneti paraméter nincs a felhasználói függvényekben. Viszont segítségükkel sokkal egyszerûbben meg lehet fogalmazni az elôbbi problémát:

```
CREATE FUNCTION osszeadas(
  @a INT,
  @b INT)
  RETURNS INT
BEGIN
  RETURN @a + @b
END

SELECT dbo.osszeadas(1,4)
```

Azért ez sokkal természetesebb, mint a tárolt eljárásos változat. De azért szedjük csak szét ízekre a függvény deklarációt! A CREATE FUNCTION jelzi, hogy ez egy felhasználói függvény lesz. Ezután jön a függvény neve. Általában a függvényeknek vannak paramétereik, ezeket zárójelben soroljuk fel a függvény neve után. A @ nem opcionális, nem esztétikai okokból raktam bele, vagy azért, mert ettôl olyan tudományos lesz, hanem azért, mert Transact SQL-ben minden változót kötelezô @-al kezdeni. A paraméter neve után meg kell adni az ô típusát. Itt majdnem az összes, a kiszolgáló által támogatott adattípust fel lehet használni, egy-két elvarázsolt image, text vagy cursor típust kivéve. A RETURNS után kell definiálni a visszatérési érték adattípusát. A kötöttségek ugyanazok, mint a paramétereknél, azaz csak „normális“ változókat használhatunk. A függvény törzsét, ahol az általunk megálmodott funkcionalitást írjuk le, a BEGIN és END kulcsszavak közé kell elhelyezni. Ennyi.
Mondja azt valaki, hogy bonyolultak a felhasználói függvények! Ha a fenti mintapélda kéznél van, minden problémát csuklóból megoldunk. Persze enyhe túlzással, és ha egy kimeneti érték elég a feladat leírásához. :)
Még egy fontos tudnivaló. A skaláris visszatérési értékû függvényekre minimum 2 tagú névvel kell hivatkozni. Azaz legalább a függvény tulajdonosát meg kell adnunk ahhoz, hogy az SQL Server felismerje a függvényünket. Ennek megfelelôen, a:

```
SELECT osszeadas(1,4)
```

hibát fog jelezni. Helyesen:

```
SELECT dbo.osszeadas(1,4) vagy
SELECT Northwind.dbo.osszeadas(1,4)
```

De mi van, ha több értéket kell visszaadnunk? Mi van, ha ráadásul azt sem tudjuk, hogy igazából hány kimeneti értékünk lesz, mert azt a tábláinkban található információk pillanatnyi állapota szabja meg? Ebben az esetben kapaszkodunk a tábla kimenetû felhasználói függvényekbe. *(A továbbiakban nem írom ki mindenhol a felhasználói jelzôt, de ott van.)*
Ezt értsük úgy, hogy, ha a skaláris függvények egy skaláris mennyiséget adnak vissza, akkor a tábla kimenetûek meg egy táblát? Igen. De, hát nincs is ilyen adattípus az SQL Server 7-ben! Abban tényleg nincs, de az SQL 2000-ben van. És nagyon szeretjük is ôket. Képzeljük el: van egy olyan változótípusunk, ami akár egy tízmillió sorból és huszonhat oszlopból álló teljes táblát el tud tárolni. Csoda, hogy szeretjük? Ez a tábla *(table)* adattípus.
Miért olyan szenzációs ez? Eddig is létre lehetett hozni átmeneti táblákat, és azokba is lehetett ideiglenes eredményeket beleírni. Persze, de a tábla adattípus felhasználásával egyrészt átláthatóbban, a természetes gondolkodáshoz közelebb álló kódot hozhatunk létre, másrészt olyan dolgokat is megvalósíthatunk, amelyeket korábban csak nagyon trükkösen vagy sehogyan sem tudtunk megtenni.
Hol használhatjuk fel a tábla kimenetû függvényeket? Minden olyan helyen, ahol eddig egy táblát adhattunk meg. Azaz leginkább a FROM záradék után.

```
SELECT cica, egér
FROM AzElsoTablaFuggvenyem('sajt')
```

Paraméterezett nézetek felhasználói függvényekkel, avagy az egy utasításból álló, tábla visszatérési értékû függvények

Mit tudtunk tenni SQL7-ben, ha azt kérték tôlünk, hogy kellene egy nézet, ami a megrendeléseket listázza ki, de úgy, hogy megadhassuk paraméterként, hogy melyik megrendelôhöz tartozó tételeket kívánjuk látni. Azaz valami ilyesmit akartunk írni:

```
CREATE VIEW OrdersByCustomer(
  @CustomerID varchar(5))
AS
SELECT * FROM Orders
WHERE
  CustomerID = @CustomerID
--Nem működik, nem fordul le!
```

Nos, ilyen nincs SQL7-ben, sôt SQL2000-ben sem! Ilyenkor jön a felmentô sereg, az egy utasításból álló, tábla visszatérési értékû függvény. Az elôbbi majdnem mûködô nézetet könnyen átalakíthatjuk egy tábla visszatérési értékû függvénnyé, ami már az elvárt funkciót valósítja meg:

```
CREATE FUNCTION OrdersByCustomer(
@CustomerID varchar(5))
RETURNS TABLE
AS
  RETURN (
  SELECT * FROM Orders
  WHERE
    CustomerID = @CustomerID)
--Teszt:
SELECT CustomerID, ShippedDate
FROM OrdersByCustomer('THEBI')
THEBI       1996-09-27
THEBI       1997-11-05
THEBI       1998-01-09
THEBI       1998-04-03
```

Mit kellett tennünk, hogy a majdnem-mûködô, de azért mégiscsak-ramaty nézetünkbôl egy jólfésült függvény legyen? A CREATE VIEW helyett CREATE FUNCTION-t írunk. Jelezzük, hogy a függvény visszatérési értéke nem holmi skalár, hanem tábla: RETURNS TABLE. Látható, hogy nem specifikáltuk az eredménytábla szerkezetét, csak egyszerûen megadtuk, hogy tábla lesz. Emiatt van, hogy az ilyen típusú függvényekben csak 1, azaz egy darab SELECT utasítás lehet, hiszen annak az eredményhalmaza határozza meg a visszatérési értékként generálódó tábla típusát. Pontosabban lehet benne egymásba ágyazva több SELECT utasítás is, de a teljes lekérdezés csak egy eredményhalmazt adhat vissza. Azaz pont ugyanaz a helyzet, mint a nézeteknél volt.

Több utasításból álló, tábla visszatérési értékû függvények

Bonyolultabb esetben a visszatérési érték nem állítható elô egyetlen SELECT utasítás segítségével, ilyenkor kell használnunk ezt a függvénytípust. Mivel ilyenkor már nem egyértelmû, hogy melyik lekérdezés kimenetét szeretnénk visszaadni, explicit deklarálnunk kell a visszatérési értékként szolgáló tábla szerkezetét egy tábla típusú változóként. A változót INSERT utasítások segítségével feltöltjük *(akárhány lépésben)*, és a RETURN utasítás ezt fogja visszaadni a hívónak. Erre a függvénytípusra összetettebb példákat a következô fejezetben találhatunk.

Praktikus felhasználói függvények

Annak örömére, hogy megkaptuk a felhasználói függvényeket, használjuk ki az alkalmat, és írjuk meg néhány olyan probléma megoldását, ami a minden napi fejlesztések során sokszor elôjött-elôjön.

Szövegelôfordulás számláló

Gyakori feladat, hogy egy szövegben meg kell keresni azt, hogy egy másik szöveg hányszor fordul elô benne. Milyen algoritmust használjunk? Az egyik legegyszerûbb, bár nem feltétlen a leghatékonyabb módszer az, hogy a keresendô szöveg minden egyes elôfordulását cseréljük ki egy üres sztringre a „nagy" szövegben *(amiben keresünk)*, és az eredeti szöveg hosszából vonjuk ki az így kapott szöveg hosszát. Ezt az eredményt már csak le kell osztani a keresendô szöveg hosszával, hisz minden csere után ennyivel csökkent az „nagy" szöveg hossza. Hogy néz ez ki függvényként? *(A bemutatott példa egy nagyon nem normalizált adatbázis, annyira nincs formában, hogy még 0. normál formában sincs. Csak demócélokra szolgál, nem adatbázistervezési minta!)*

```
CREATE FUNCTION StringOccur
(
  @cString  AS varchar(8000),
  @cLookFor AS varchar(100)
)
RETURNS int
AS
BEGIN
  RETURN
  (LEN(@cString)
  -LEN(REPLACE(@cString, @cLookFor, '')))
  / LEN(@cLookFor)
END
--Teszt tábla
CREATE TABLE T1
(
  cMenu varchar(100) NOT NULL
)
--Tesztadatok
INSERT INTO T1 VALUES('Töltött káposzta, Almáspi
↳ te, Diósbejgli')
INSERT INTO T1 VALUES('Pulykarizottó, Mákosbejgli,
↳ Diósbejgli')
INSERT INTO T1 VALUES('Székelykáposzta, Rántottbé
↳ ka, Mákosbejgli')
INSERT INTO T1 VALUES('Stefániasült, Káposztáspi
↳ te, Túrósbejgli')

SELECT
  cMenü AS Menü,
  dbo.StringOccur(cMenu, 'káp') AS
  Káposztásfogás,
  dbo.StringOccur(cMenu, 'bejgli') AS
  Bejglitartalom,
  dbo.StringOccur(cMenu, 'Mákos') AS
  Mákosfogás
FROM T1
```

A kimenet *(nyomdai okokból táblázatban)*:

Menü			Káposztás fogás	Bejgli tartalom	Mákos fogás
Töltött-káposzta	Almás-pite	Diós-bejgli	1	1	0
Pulyka-rizottó	Mákos-bejgli	Diós-bejgli	0	2	1
Székely-káposzta	Rántott-béka	Mákos-bejgli	1	1	1
Stefánia-sült	Káposztás-pite	Túrós-bejgli	1	1	0

A függvény elég trükkös, megér néhány szót. „Izomból" nekifutva hogyan oldanánk meg a példát? Egy ciklusban keresnénk a keresendô szöveg elôfordulásait a „nagy" szö-

vegben, mindig a következô pozíción *(karakteren)* folytatva a „nagy" szövegben, mint ahol az elôzô lépésben abbahagytuk. Ehhez a megoldáshoz ciklust kellene szerveznünk, ami jelentôsen megbonyolítaná a megoldást. Ehhez képest a fenti függvény sokkal egyszerûbb, hisz a bonyolultabb funkcionalitást átadtuk a REPLACE függvénynek.
Más kérdés, hogy az imént vázolt algoritmus és a fenti algoritmus más kimenetet ad például a következô szövegekre:

```
-A fenti függvény (a LEN-es)
SELECT dbo.StringOccur('babababababa', 'baba')
2
```

Ezzel ellentétben, ha lenne egy függvényünk, ami az imént említett módon mûködne, akkor a visszaadott érték 4 lenne, hisz:

babaababa
ba**baba**baba
baba**baba**ba
bababa**baba**

A kérdés az, hogy átlapolhatják-e egymást a keresendô szöveg elôfordulások? Ha nem, akkor jó a fenti függvény, ha igen, akkor meg kell írni a másik verziót. Ezt a konkrét feladat határozza meg.

Szövegdarabolás

Visual Basic programozók gyakran keresik a Basic Split függvény Transact SQL párját. Mindhiába, mert nincs. A Split egy nagyon hasznos függvény, arra való, hogy egy szöveget valamilyen határoló karakter mentén feldaraboljon, és a darabokat visszaadja egy tömbben. Segítségével egy mondatot feldarabolhatunk szavakra, egy vesszôvel elválasztott listát listaelemekre, satöbbi.
Mivel nincs ilyen függvényünk, implementáljunk egyet! Az elsô akadály, amibe rögtön beleütközünk az, hogy a TSQL-ben nincs tömb típus. Emiatt a függvény kimenete tábla típusú kell, hogy legyen, mert skalárban nem tudunk visszaadni több elemet. Azaz, írjunk egy olyan függvényt, ami a megadott szöveg és az elválasztó karakter ismeretében szétdarabolja a szöveget, és egy táblában visszaadja a szövegkomponenseket. Legyen a visszaadott mezô neve cStringPart!

```
CREATE FUNCTION Split
(
  @cOriginalString AS varchar(8000),
  @cDelimiter char(1))
  RETURNS @SplitString table
(
  nID int IDENTITY(1,1) NOT NULL,
  cStringPart varchar(8000) NULL)
AS
BEGIN
  DECLARE @nNumberOfDelimiters AS int
  --Számoljuk  meg, hány határoló
  --karakterünk van.
  --Használjuk fel az elôzôleg megírt
  --szöveg-elôfordulás számláló
  --függvényünket.
  SET @nNumberOfDelimiters =
  dbo.StringOccur(@cOriginalString, @cDelimiter)
  DECLARE @i AS int
  SET @i = 0
  --Végigmegyünk az összes szövegdarabon
  WHILE @i < @nNumberOfDelimiters
  BEGIN
  --A forrásszöveg baloldalából
  --kivágjuk az ott található szöveget
  --a határoló karakterig,
  --és beszúrjuk az eredménytáblába.
    INSERT INTO
      @SplitString
    SELECT
      LEFT(@cOriginalString,
      CHARINDEX(@cDelimiter,
      @cOriginalString)-1)
    --Levágjuk a már feldolgozott
    --szöveget, így az elején mindig
    --megtaláljuk a következô darabot.
    SET @cOriginalString =
    SUBSTRING(@cOriginalString,
    CHARINDEX(@cDelimiter,
    @cOriginalString)+1, 8000)
    --Továbblépünk a következô darabra

    SET @i = @i + 1
  END
  --Az utolsó határoló karakter után
  --még maradt egy darab, azt is
  --szúrjuk be az eredményhalmazba.
  INSERT INTO
    @SplitString
  VALUES
    (@cOriginalString)
  --Összeállt az eredménytábla, ideje
  --visszaadni azt a hívónak.
  --Itt már nem kell jelezni, hogy mit
  --adunk vissza, mert az már a RETURNS-
  --nél (az elején) megtettük.
  RETURN
END
--Teszt

SELECT
  T1.*
FROM
  Split('Dec 24,Dec 25,Dec 26,Dec 31',',')
  AS T1
--Eredmény:

nID          cStringPart
1            Dec 24
2            Dec 25
3            Dec 26
4            Dec 31
```

Látható, hogy a függvények egymásba ágyazhatók, éppúgy, mint a tárolt eljárások. Ezzel élve nagyon jól átlátható, moduláris programokat írhatunk az SQL Server-re.

Spam-re fel!

Az SQL Server segítségével könnyedén írhatunk körlevele-

ket, ha van egy címzett *(áldozat)* adatbázisunk. A klasszikus megoldásban kurzort használnánk, és az xp_sendmail külsô tárolt eljárást hívnánk meg egy ciklusban. Azonban a kurzorok használata elég körülményes dolog. Keressünk egy jóval egyszerûbb megoldást, természetesen a felhasználói függvények felhasználásával! A megcélzott függvény célja egyszerû: a bemenô paraméterekben meghatározott címzettnek elküldeni egy E-mail-t.

```
--Létrehozzuk a függvényt
CREATE FUNCTION SendMail
(
  @cRecepients AS varchar(200),
  @cSubject AS nvarchar(100),
  @cBody AS nvarchar(3000)
)
  RETURNS INT
BEGIN
  DECLARE @nResultCode INT
  EXEC @nResultCode = master..xp_sendmail
  @recipients = @cRecepients,
  @subject = @cSubject,
  @message = @cBody
  RETURN @nResultCode
END
--Egy teszttábla a „célszemélyekhez"
CREATE TABLE SpamTarget
(
  nID int NOT NULL IDENTITY(1,1),
  cTargetEmail nvarchar(400) NOT NULL,
  cFirstName nvarchar(100) NOT NULL,
  cLastName nvarchar(100) NOT NULL
)
--Két áldozat felvitele

INSERT SpamTarget VALUES
('Zsolt.Soczo@w2ktest1.vodafone.hu', 'Zsolt', 'Soczó')
INSERT  SpamTarget  VALUES  ('ECudar@vadmalac.hu',
'Elek', 'Cudar')
--A levelek elküldése.
SELECT
  dbo.SendMail(cTargetEmail,
  cFirstName +
  '! Nyerj 99999999999 Forintot!',
  'Legyél te is milliomos!')
FROM
  SpamTarget
--Az áldozat által kapott levél:
From: sqlacc
Sent: Saturday, December 23, 2000 6:08 PM
To: Zsolt Soczo
Subject: Zsolt! Nyerj 99999999999 Forintot!
Legyél te is milliomos!
```

Anti-hacking toolkit v0.0

Utolsó és egyben legbonyolultabb függvényünkben egy állomány épség *(eredetiség)* ellenôrzô programot írunk. A dupla KV ismét javasolt elôtte, mert elég bonyolult lesz.
A feladat, hogy dolgozzunk ki egy olyan módszert, amely segítségével a védendô állományok bizonyos jellemzôit letároljuk, majd egy ellenôrzô rutint lefuttatva ellenôrizzük, hogy a jellemzô azonos-e a letárolt, háborítatlan értékkel. Ha nem, akkor a megfigyelt állományt egy rosszindulatú hacker vagy egy még rosszabb indulatú telepítôprogram módosította. A példa kedvéért az állomány méretet használjuk fel az ellenôrzéshez. Ennél sokkal profibb megoldás lenne, ha az állományokhoz kiszámítanánk valamilyen ellenôrzô értéket *(pl. MD5 hash)*, és ezt tároljuk le az adatbázisban. Így sokkal nagyobb valószínûséggel lehetne jelezni, hogy megváltozott egy állomány.
Hogyan látnánk neki a feladat megoldásának? Mivel a fájlok méretét közvetlenül nem lehet lekérdezni az SQL Serverbôl, kénytelenek vagyunk kinyúlni a szerverbôl. Ehhez valamilyen külsô tárolt eljárásra lesz szükségünk. Az **xp_cmdshell**, amivel külsô parancsokat lehet végrehajtani, szinte kínálja magát, hogy bevessük erre a feladatra. Meghívunk egy VBScript programot, ami visszaadja a paraméterként megadott állomány hosszát. A külsô parancs futtatásából származó sorokat, azaz a fájl hosszát az **xp_cmdshell** táblaként adja vissza, aminek az elsô sora tartalmazza a kívánt eredményt. Hogyan nyerjük ki ebbôl a táblából az elsô sort? Próbáljuk belerakni egy átmeneti *(temporary)* táblába, és abból leválogatni az eredményt. Ez azonban sajnos nem megy, mert függvényben nem használhatunk temporary táblát.
Próbáljuk meg belemásolni az **xp_cmdshell** kimenetét egy table típusú változóba. Ez sem megy, mert az **INSERT Tábla (EXECUTE xp_cmdshell...)** típusú parancs, *(ami egy tárolt eljárás kimenetét beszúrja egy táblába)* nem megy tábla típusú változóval, csak valódi táblával. „Valódi" táblát viszont nem módosíthat egy függvény. Van ebbôl kiút?
Utolsó kapaszkodóként elfeledkezünk az **xp_cmdshell**-rôl, és megpróbáljuk felhasználni a külsô COM komponensek meghívására szolgáló függvényeket. És ez bejön! A FileSystemObject COM komponens közvetlen meghívásával célba érünk. A kódhoz tartozó magyarázatot beleszôttem a kódba, mert kiragadva kevésbé érthetô lenne.

```
--A fájlméret lekérdező függvény
--deklarációja.
CREATE FUNCTION GetFileSize
(
  @cFilePath AS nvarchar(4000)
)
 RETURNS INT
BEGIN
  DECLARE @nFileSize int
  DECLARE @hr int
  DECLARE @objFileSystem int
  DECLARE @objFile int
--Hozzunk létre a FileSystemObject-
--ből egy példányt.
  EXEC @hr = sp_OACreate
  'Scripting.FileSystemObject',
  @objFileSystem OUT
--Ha hiba történt egyszerűen visszatérünk
--egy hibakóddal. Ez azért nagyon csúnya,
--mert a kód későbbi részeiben
--bekövetkezett hiba esetén
--felszabadítatlan objektumok maradnak a
--memóriában!
```

```
--Így produkciós környezetben le
--kell kezelni a hibákat megfelelő módon.
--Ehhez az sp_OAGetErrorInfo külső tárolt
--eljárást lehet segítségül hívni.
  IF @hr <> 0 RETURN –1
--Meghívjuk a FileSystemObject GetFile
--nevű metódusát, ami visszatér egy
--File típusú objektummal. Ezt az
--@objFile változóban tároljuk el.
--A hívás paramétere az a fájlnév, aminek
--keressük a méretét (@cFilePath).
  EXEC @hr = sp_OAMethod @objFileSystem,
  'GetFile', @objFile OUT, @cFilePath
  IF @hr <> 0 RETURN –1
--A File objektum Size nevű
--tulajdonságának lekérdezésével
--megkapjuk a keresett állomány méretét.
--A kapott szám az @nFileSize-ba kerül.
  EXEC @hr = sp_OAGetProperty @objFile,
  'Size', @nFileSize OUT
  IF @hr <> 0 RETURN –1
--Felszabadítjuk a létrehozott
-- objektumokat.
  EXEC @hr = sp_OADestroy @objFile
  IF @hr <> 0 RETURN –1
  EXEC @hr = sp_OADestroy @objFileSystem
  IF @hr <> 0 RETURN –1
--Visszatérünk a kapott értékkel.
  RETURN @nFileSize
END
--Ebben a táblában tároljuk a fájlokat,
--és a hozzájuk tartozó méreteket.
CREATE TABLE FileAuthority
(
  nID int NOT NULL IDENTITY(1,1),
  cFileName nvarchar(3000) NOT NULL,
  nFileSize int NOT NULL
)
--Néhány tesztállomány. A –2-vel jelezzük
--hogy még soha nem olvastuk ki az adott
--fájl hosszát.
INSERT FileAuthority VALUES
('c:\winnt\notepad.exe', -2)
INSERT FileAuthority VALUES
('c:\boot.ini', -2)
INSERT FileAuthority VALUES
('c:\ntldr', -2)
INSERT FileAuthority VALUES
('c:\io.sys', -2)
INSERT FileAuthority VALUES
('c:\ntdetect.com', -2)
--Ezzel a tárolt eljárással
--töltjük fel a táblázat méret mezőit.
CREATE PROCEDURE CalculateFileSize
AS
  UPDATE
    FileAuthority
  SET
    nFileSize = dbo.GetFileSize(cFileName)
--Futtassuk le. Ettől kezdve van egy
--táblázatunk arról, hogy melyik fájlnak
--milyen hosszúnak kell lenni.
EXEC CalculateFileSize
--Nézzük meg, mit tartalmaz a táblánk!
--SELECT * FROM FileAuthority
FileName                          FileSize
c:\winnt\notepad.exe              50960
c:\boot.ini                       195
c:\ntldr                          214416
c:\io.sys                         0
c:\ntdetect.com                   34468
--Ezzel a tárolt eljárással össze
--lehet hasonlítani a letárolt és
--a futtatás pillanatában aktuális
--állományhosszakat.
--Csak azokat listázza ki, amelyeknél
--eltérés van a két érték között.
CREATE PROCEDURE CheckFileSize
AS
  SELECT
    nID,
    cFileName,
    nFileSize AS OriginalSize,
    dbo.GetFileSize(cFileName) AS
    CurrentSize
  FROM
    FileAuthority
  WHERE
    nFileSize <>
    dbo.GetFileSize(cFileName)
--Tesztképpen megváltoztattam a boot.ini
--fájl hosszát.
--Ellenőrizzük le!
EXEC CheckFileSize

A kimenet:
nID cFileName   OriginalSize CurrentSize
2   c:\boot.ini 195          197
```

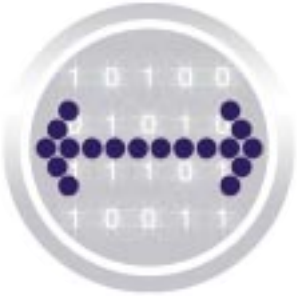
Hoppá, a BOOT.INI-t valaki megváltoztatta! Mûködik az ellenôrzô eljárásunk.

Zárszó

A cikkben felhozott példák két igen fontos dologra világítanak rá. A felhasználói függvények felhasználásával nagyon sok, a gyakorlatban felmerülô feladatot oldhatunk meg, amelyeket eddig csak átmeneti táblák és kurzorok felhasználásával tudtunk megtenni, általában nagyon bonyolultan, és nehezen olvasható módon. A másik tanulság, hogy a külsô tárolt eljárások segítségével sok olyan feladatot is megoldhatunk az SQL Server segítségével, amelyeket általában más programnyelven megírt programokkal *(Visual Basic, stb.)* végeztettünk el.
A következô részben a kurzorokról lebbentem fel a fátylat, megnézzük, hogy körültekintô felhasználásukkal a felhasználói függvényekhez hasonlóan igen bonyolult feladatokat is elég egyszerûen megoldhatunk.

Soczó Zsolt
MCSE, MCSD, MCDBA
Protomix Rt.

Transact SQL (V. rész)



Bevezetés

Sok haladó SQL szerver programozó számára is ismeretlen a kurzorok fogalma. Pedig helyes használatuk alapvetôen meghatározza alkalmazásaink teljesítôképességét, különösen sokfelhasználós környezetben. Mit is tudnak ôk? Mire valók? A cikk választ ad ezekre a kérdésekre. Sôt továbbmegyek. A cikk második felében kilépünk a tiszta SQL világból a való világba, és átmegyünk ADO, VB programozóba. Megtanuljuk, hogyan kell százmillió rekordból ezred másodpercek alatt leválogatni az elsô pár száz sort. Semmi TOP 100, semmi varázslás, csak kurzorok. Meglepô lesz, tartsanak velem.

Mi az a kurzor?

Természetesen mindenki tudja a választ. Az a kis bigyó, ami a szövegszerkesztôben a sor végén villog, és kijelöli, hogy a következô karakter hová fog kerülni, ha leütjük a billentyûzet valamely gombját. És ez nem is áll messze az SQL Server kurzorától!

Az SQL Server halmazokban gondolkodik. Amikor végrehajtunk egy lekérdezést, akkor annak van egy kimenete, ami egy n darab sorból álló halmaz, amit eredményhalmaznak (result set) hívunk. Amikor a WHERE feltétellel szûrjük az adatokat, akkor halmazmûveleteket végzünk. Pl. WHERE Fizetés > 100, akkor arra kérjük a kiszolgálót, hogy a teljes halmazból *(ami például egy teljes tábla tartalma)* csak azokat az elemeket tartsa meg, amelyekre teljesül a megadott feltétel. Nem mondjuk azt SQL-ben, hogy:

```
for menjünk végig az összes soron a táblában
  ha a Fizetés > 100, akkor rakjuk bele a
  ↳ kimenetbe
next
```

Azért nem kell leírnunk ilyeneket, mert az SQL nyelv halmazokban gondolkodik, így elég a fenti WHERE kifejezés is. Ez nagyon jól van így, mert a lényegre tudunk koncentrálni, nem kell ciklusszervezéssel bajlódnunk. Azonban az élet nem ilyen egyszerû. Rengeteg helyen nem halmazokra, hanem egyedi rekordokra van szükségünk, amelyeken egy ciklusból végigszaladunk, és valamilyen mûveletet végzünk a segítségükkel vagy rajtuk. Például írunk egy alkalmazást, ami kilistázza az ügyfelek tábla tartalmát. Megjelenítjük az ügyfelek nevét egy listában, és ha a program kezelôje kiválaszt egy nevet, a lista alatt megjelenik az ügyfél összes adata. Ha akarja, módosíthatja ezeket, majd a listában rákattint a következô ügyfélre, és azzal foglalkozik. Azért ez szekvenciális feldolgozás nem? Ez teljesen eltér az SQL Server halmazos lelkivilágától. Szaknyelven ezt a felfogáskülönbséget impedancia különbözôségnek hívjuk *(több ilyet nem mondok, ígérem :)*.

Az SQL Server teljességgel használhatatlan lenne, ha nem oldaná fel a két világ közötti ellentétet, az egyedi rekordokkal foglalkozó való világ és a tiszta SQL-es halmazokból építkezô világ között. Erre lesznek jók a kurzorok! Ôk a kapu, az átjáró a két világ között.

Képzeljünk el egy felhôt, amiben kis, masnis, sorszámozott csomagocskák lebegnek. Ez az SQL eredményhalmaz. Most képzeljük el, hogy jön egy óriás, aki egymásra rakja a dobozkákat, szépen sorban. Azt mondjuk neki: „Te, Küklopsz, add ide nekem a legfelsô doboz tartalmát!" És ô odaadja nekünk a dobozban található dolgokat, információkat, egy rekord tartalmát. Majd, hadd mozogjon, azt mondjuk neki, hogy „Kükkancs, most nekem a legutolsó kellene!", és már a miénk is a legalsó dobozka tartalma. Sôt, ez a behemót olyan okos, hogy azt is megérti: „Nagyfiú a legutóbbi elôtt öttel levô doboz tartalma kellene!", és már adja is az alulról hatodik dobozka tartalmát.

A kurzor pont olyan okos, mint Küki. Ha egy SQL eredményhalmazra nyitunk egy kurzort, akkor a kiszolgáló kivasalja az eredményhalmazt, és készít belôle egy hurkát, amin végig lehet gyalogolni. Lehet neki olyan parancsokat adni, mint „kettôvel elôre", vagy „a legelsôt" satöbbi. És természetesen az aktuális pozícióban található sor *(rekord)* adatait ki lehet olvasni, sôt lehet módosítani is.

Másképpen fogalmazva a kurzor egy olyan nevesített eredményhalmaz, amiben a kiszolgáló mindig nyilvántartja az aktuális pozíciót, és amiben léptethetjük azt elôre-hátra. Olyan ez, mint amikor a telefonkönyvben keresünk valakit. Az ujjunk mindig rajta áll azon a soron, amit éppen olvasunk. A telefonkönyv az eredményhalmaz, az ujjunk pedig a kurzor, ami mindig egy bizonyos rekordra mutat.

Az SQL Serverrel kétféle kurzort használhatunk, és a kettô közötti különbségtétel nagyon fontos! Az egyik típus az, amit Transact SQL-ben, általában tárolt eljárásokban, triggerekben használunk, és a DECLARE CURSOR kulcsszóval hozunk létre. Ezek akkor jók, ha az SQL programunkban sorról-sorra kell valamilyen mûveletet végezni, olyat, amit a hagyományos SELECT, UPDATE stb. utasításokkal nem lehet megfogalmazni. A cikk elsô fele ezekkel fog foglalkozni. A másik fajta az, amit az adatbázismeghajtó programok *(manapság zömében OLE-DB-n, ADO-n keresztül)* valósítanak meg. Ezek elônye, hogy eltakarják a „piszkos részleteket", viszont csak valamilyen egyéb nyelven és eszközzel *(VB, VC, stb.)* lehet használni. Cikkünk második felében az ilyen típusú kurzorokkal foglalkozunk.

Egy egyszerû kurzor létrehozása

Annyit már tudunk a kurzorokról, hogy kell hozzájuk valamilyen lekérdezés, ami egy eredményhalmazt generál, és ehhez lehet valamilyen módon hozzáilleszteni egy kurzort, amivel azután szabadon mozoghatunk a leválogatott rekordok között. Nézzük meg részleteiben, hogyan néz ki ez a folyamat.

1. Deklaráljuk a kurzort a DECLARE utasítás segítségével. Ez már ismerôs lehet, hiszen a változókat is ezzel lehetett létrehozni. A különbség az, hogy a kurzor nevében nem

használhatunk @-ot, ellentétben a változókkal, ahol meg kötelezô odarakni. A deklaráció során kétféle viselkedését lehet specifikálni, az egyik azt szabja meg, hogy a kurzor által bejárni kívánt recordset mennyire tükrözze az eredeti adathalmazban bekövetkezô változtatásokat, a másik pedig azt, hogy milyen irányokban *(elôre-hátra, csak elôre)* lehet bejárni a kapott eredményhalmazt. Itt kell megadni azt a SELECT utasítást is, aminek az eredményhalmaza táplálja a kurzort. Mivel a kurzorral bejárt rekordok már egy rendezett halmazt alkotnak, azért sokszor van értelme használni az ORDER BY-t is a rekordok rendezésére. Az eddigieknek megfelelô leegyszerûsített példakód így néz ki:

```
DECLARE curTest CURSOR
FOR
SELECT
  EmployeeID, LastName, FirstName
FROM
  Employees
```

2. Megnyitjuk a kurzort. A legegyszerûbb ezt úgy felfogni, mint hogy végrehajtjuk a deklarációban kijelölt lekérdezést.

```
OPEN curTest
```

3. Mozgatjuk, pozícionáljuk a kurzort a megfelelô bejegyzésre, és felhasználjuk az aktuális pozíción található sorok tartalmát. A mozgatásokat a FETCH utasítás segítségével tehetjük meg. A FETCH FIRST ráállítja a kurzort az elsô rekordra. Ennek megfelelôen a FETCH LAST az utolsóra. A következô rekordot a FETCH NEXT-tel érhetjük el. Mivel általában rekordról-rekordra lépkedünk, ezt az utasítást használjuk leggyakrabban. Aki szeret visszafele bejárni egy eredményhalmazt, annak jól jön a FETCH PRIOR utasítás, ami az elôzô sorra lép vissza. Valamivel izgalmasabb a FETCH ABSOULTE n és a FETCH RELATIVE n. A nevükbôl elég jól kiviláglik a feladatuk. A FETCH ABSOULTE n a kisimított eredményhalmaz n. rekordjára pozícionálja rá a kurzort, függetlenül attól, hogy hol állt éppen. A FETCH RELATIVE n pedig az aktuális pozíciótól tudja n távolságra elmozgatni a kurzort. Ezt a két utasítást nem minden típusú kurzorra lehet kiadni, de ezekrôl majd egy kicsit késôbb.

```
FETCH NEXT FROM curTest
```

4. Amellett, hogy barangolunk a kurzorral és kiolvassuk az általa kijelölt sor tartalmát, még módosíthatjuk és törölhetjük is az adott sort. Ezeket a lehetôségeket ritkán használjuk ki, de azért jól jöhetnek még. Az érdekességük ezeknek a speciális UPDATE és DELETE utasításoknak, hogy a WHERE feltételben nem egy „hagyományos” sort kiválasztó feltételt adunk meg, hanem a CURRENT OF kurzornév kifejezést, amiben a kurzornév az általunk definiált kurzort reprezentálja. Azaz a módosító és törlô utasítások valahogy így néznek ki:

```
UPDATE tábla SET ... WHERE CURRENT OF kurzornév
DELETE tábla          WHERE CURRENT OF kurzornév
```

5. A Mór megtette a kötelességét, lezárjuk a kurzort. A késôbbiekben újra megnyithatjuk anélkül, hogy újra deklarálnánk, de már nem olvashatunk ki rajra keresztül rekordokat, illetve nem pozícionálhatjuk a kurzort.

```
CLOSE curTest
```

6. Felszabadítjuk a kurzort. Miután megnézzük a kurzorok típusait, látni fogjuk, hogy a kurzorunk által bejárni kívánt eredményhalmaz fenntartása igen sok kiszolgálóoldali erôforrást köthet le. Emiatt érdemes azonnal felszabadítani, amint felhasználtuk az eredményeket.

```
DEALLOCATE curTest
```

A bejárás zegzugos részletei

Láttuk, hogy a FETCH utasítás variánsaival keresztül-kasul bejárhatjuk az eredményhalmazt. De honnak tudjuk, hogy a végére értünk? És hogy már az elején járunk? És azt honnan vesszük észre, hogy az éppen kiolvasni kívánt sort valaki más már törölte alólunk? Nos, ezekre a státuszinformációk kiolvasására hozták létre a @@FETCH_STATUS nevû függvényt (globális változót, ki hogy szereti). Ha a @@FETCH_STATUS értéke 0, akkor a megelôzô FETCH utasítás sikeresen hajtódott végre, és felhasználhatjuk a kurzor által kijelölt rekordot. Ha -1-et kapunk vissza, akkor túlszaladtunk az eredményhalmazon, azaz vagy a legelsô sor elé akartunk menni egy FETCH PRIOR-ral, vagy a legutolsón túlra a FETCH NEXT-tel. Ha ciklusban járjuk végig a sorokat, akkor erre szoktuk építeni a ciklus végét jelzô feltételt. A -2 a legravaszabb. Ez vagy azért jön elô, mert az aktuális sort törölték, vagy pedig azért, mert úgy módosították az adott sort, hogy az már nem esik bele abba az eredményhalmazba, amit a deklarációnál használt SELECT jelöl ki. Például a SELECT-tel kiválasztjuk a budapesti alkalmazottakat, és az így leválogatott halmazban barangolunk a kurzorunkkal. Eközben Mariska a HR-rôl átírja Nagy Elek lakcímét Szegedre. És mi a következô lépésben *(FETCH NEXT)* ki akarjuk olvasni Nagy Elek adatait, és kapunk egy nagy -2-t mert Elek már nem budapesti, így nem is lehet benne a kiinduló SELECT által leválogatott halmazban. Lássunk hát egy olyan példát, amiben végigmegyünk az összes alkalmazotton egy kurzorral:

```
DECLARE curTest CURSOR
FOR
SELECT
  EmployeeID, LastName, FirstName
FROM
  Employees
OPEN curTest
FETCH NEXT FROM curTest

WHILE @@FETCH_STATUS = 0
BEGIN
  FETCH NEXT FROM curTest
END

CLOSE curTest
DEALLOCATE curTest
```

Ez így nagyon szép, de mi történik a FETCH NEXT-ek során kiválasztott sorokkal? Ha a Query Analyser-ben kipróbáljuk az elôbbi példát, akkor az alábbi kimenetet kapjuk:

```
EmployeeID  LastName             FirstName
----------- -------------------- ----------
1            Davolio              Nancy

(1 row(s) affected)

EmployeeID  LastName             FirstName
----------- -------------------- ----------
2            Fuller               Andrew

(1 row(s) affected)
...
```

Azaz minden egyes FETCH NEXT egy új eredményhalmazt generál! Hát ez minden, csak nem álom *(hacsak nem rémálom)* feldolgozni egy ügyfélalkalmazásból, arról nem is beszélve, hogy minden egyes FETCH NEXT eredményeképpen elôállt eredményhalmaz egyenként át kell, hogy utazzon a hálózaton. Egy sima SELECT összes sora egy nagy csomagban *(itt nem hálózati csomagról, hanem körülfordulásról van szó)* utazik, míg a kurzor minden egyes sora külön csomagban. Ez aztán az erôforráspazarlás netovábbja. Általában nem is használjuk így a kurzorokat, legalábbis a Transact SQL kurzorokat. Hisz milyen elônyét élveztük annak, hogy a

```
SELECT
  EmployeeID, LastName, FirstName
FROM
  Employees
```

helyett egy jó bonyolult kódot hordtunk össze? Semmit. Ilyen módon nem jól hasznosíthatók a kurzorok. Azonban a bejárás során érintett sorokat elraktározhatjuk változókba is, és ekkor lesz csak igazán nagy az örömünk. Nulla darab eredményhalmaz generálódik, a hálózaton csönd lesz, és mi hatékonyan, gyorsan dolgozunk kurzorunkkal a kiszolgálóoldalon. Hogyan is?
Deklarálunk lokális változókat, ezekbe rakjuk az aktuálisan felolvasott rekord mezôinek a tartalmát:

```
DECLARE @nEmployeeID INT
DECLARE @cLastName VARCHAR(20)
DECLARE @cFirstName VARCHAR(20)
```

A FETCH ... utasítás után elhelyezünk egy INTO kulcsszót, és felsoroljuk az elôbbi változóinkat, amelyekbe szeretnénk belerakni a kurzor által „kiolvasott” rekord tartalmát:

```
FETCH NEXT FROM
  curTest
INTO
  @nEmployeeID, @cLastName, @cFirstName
```

A felsorolt változók sorrendje meg kell, hogy egyezzen a SELECT által generált eredményhalmaz elemeinek a sorrendjével, hogy egymásra találjanak az értékek.

Hogy kerek legyen a példánk, írjunk egy olyan kódot, ami összegyúrja egy nagy listává az összes alkalmazott nevét, azaz összefûzi ôket egy sztringgé. Ezt elég nehéz, ha egyáltalán lehetséges megoldani „hagyományos” SELECT felhasználásával:

```
DECLARE @nEmployeeID INT
DECLARE @cLastName VARCHAR(20)
DECLARE @cFirstName VARCHAR(20)
DECLARE @cAllNames VARCHAR(4000)
SET @cAllNames = ''

DECLARE curTest CURSOR
FOR
SELECT
  EmployeeID, LastName, FirstName
FROM
  Employees
OPEN curTest

FETCH NEXT FROM
  curTest
INTO
  @nEmployeeID, @cLastName, @cFirstName

WHILE @@FETCH_STATUS = 0
BEGIN
  SET @cAllNames = @cAllNames +
      @cLastName + ' ' +  @cFirstName + ', '

  FETCH NEXT FROM
    curTest
  INTO
    @nEmployeeID, @cLastName, @cFirstName
END

CLOSE curTest
DEALLOCATE curTest
--Teszt:
SELECT @cAllNames AS Names
```

Az utolsó teszt SELECT eredményeként láthatjuk, hogy a @cAllNames tartalma:

```
Davolio Nancy, Fuller Andrew, Leverling Janet,
Peacock Margaret, Buchanan Steven, ...
```

Kurzortípusok

Mint korábban említettem, többféle kurzort hozhatunk létre, annak megfelelôen, hogy mi a célunk a keletkezô eredményhalmazzal. Nézzük végig a lehetôségeket, és azt, hogy mikor melyikkel érdemes élni.

Statikus kurzorok

Ha a kurzor deklarációja során a statikus típust kérjük, akkor az SQL Server végrehajtja a kért lekérdezést, és a teljes eredményhalmazt elhelyezi a tempdb-ben, egy ideiglenes táblában. Azaz kapunk egy másolatot a lekérdezés eredményébôl, így a megnyitás után akár le is törölhetik az összes sort a kiinduló táblából, mi vidáman lépkedünk a másolaton. Azaz ennél a kurzornál soha nem lesz a @@FETCH_STATUS értéke –2, nem lapátolják ki alólunk a

sorokat. Más kérdés, hogy nem is fogjuk fel, hogy közben elszaladt mellettünk a világ. Mivel másolaton dolgozunk, az aktuális sort nem is módosíthatjuk, azaz a korábban ismertetett UPDATE, DELETE nem használható erre a kurzorra. Deklaráció:

```
DECLARE Kurzornév CURSOR STATIC
FOR ...
```

Mikor jó egy statikus kurzor? Hmm. Igazából nem tudok jó felhasználást. Ha minden sorra szükségünk van, akkor kár az egész táblát átmásoltatni a tempdb-be, egyszerûbb lekérdezni a teljes eredményhalmazt a hagyományos módon, mindenféle kurzor felhasználását mellôzve *(ezt hívják Default Result Set-nek)*. Ne nagyon használjuk a statikus kurzort, csak ha valami különleges indokunk van rá.

Keyset kurzorok

A keyset kurzor már sokkal gazdaságosabban bánik a tempdb-vel, mint a statikus kurzor. Nem a kurzordeklarációban elôírt teljes sorokat tárolja le egy átmeneti táblába, csak az egyes sorok egyedi azonosítóit. Ebbôl következôen a lekérdezésben szereplô táblának kell lennie legalább egy olyan oszlopának, ami garantáltan azonosítja a sorokat, garantáltan egyedi. Ez lehet egy UNIQUE index, egy PRIMARY KEY vagy egy clustered index *(lehet, hogy a clustered index értékei nem egyediek, de az SQL Server minden clustered index bejegyzéshez hozzácsatol egy belsô azonosítót, amitôl azok belülrôl egyediek lesznek)*.

Mivel csak a kulcsokat tároljuk a tempdb-ben, a kiválasztott valódi sorokat lehet, hogy valaki más módosította a kurzor megnyitása óta. Ilyenkor a megváltozott mezô értékeket kapjuk vissza. Ha valamelyik sort közben törölték, akkor a @@FETCH_STATUS –2 jelzi, hogy már nem létezik a sor, amit ki szeretnénk olvasni. Ha olyan értékeket szúrnak be a táblába, aminek a deklarációban szereplô SELECT alapján benne kellene lennie a kurzor eredményhalmazában, nos, azokat nem fogjuk látni. A legenerált kulcshalmaz fix, nem változik, csak ha lezárjuk, és újra megnyitjuk a kurzort. Formátum:

```
DECLARE curTest CURSOR KEYSET
FOR ...
```

Mikor érdemes használni ezt a kurzort? Akkor, ha csak a kiválasztott sorokkal szeretnénk foglalkozni, és nem érdekel bennünket az, hogy esetleg közben további sorokat szúrtak be egy táblába, de érdekel bennünket a kiválasztott sorokat érintô változtatási kísérlet.

Dinamikus kurzorok

Láttuk, hogy a statikus kurzornak mindegy mi történik a kiinduló adatokkal, mi vígan olvassuk a legenerált másolat eredményhalmazt. A keyset kurzor már figyelmesebb, a sorokon végzett UPDATE és DELETE utasításokat már visszatükrözi a kurzort használó alkalmazásnak. De az INSERT-eket nem, azaz nem jelennek meg a kurzor megnyitása után beszúrt sorok a kurzor eredményhalmazában. Érezzük, hogy már csak egy lépés van hátra egy olyan kurzortípushoz, ami az összes változtatást átvetíti a kurzor eredményhalmazára. Természetesen ez a dinamikus kurzor. Ez nem nyúl a tempdb-hez *(legalábbis nem jelentôs mértékben)*. Nem hoz létre átmeneti táblákat. A kurzorral sétáló alkalmazás észreveszi, ha új sort szúrnak be a táblába, mert az új sorok megjelennek a kurzor eredményhalmazában! Nem véletlenül dinamikus a neve. A deklaráció nem fog meglepetést okozni:

```
DECLARE curTest CURSOR DYNAMIC
```

Valójában elég nehéz elképzelni, hogy az SQL Server tervezôi hogyan valósították meg ezt a funkcionalitást. Biztos nem volt egyszerû. De sikerült nekik, így van egy szuperjó kurzorunk, ami nagyon kicsi kiszolgálóoldali terhelést okoz. Akkor jön igazán jól, ha egy lekérdezés kimenete nagyon nagy eredményhalmazt adna vissza, de nekünk ebbôl csak az elsô, például 50 sorra van szükségünk. Az SQL guruk ilyenkor TOP 50-ért kiáltanak, amivel kapásból egy 50 elemûre vágott eredményhalmazt kapunk. De mi van, ha nekünk csak az 550. és a 600. sor közötti rekordok kellenek? A TOP 600-zal leválogatom az elsô hatszáz eredményt *(ami azt jelenti, hogy azok át is csurognak a hálózaton az ügyfélprogramra, a modemesek nagy örömére)*, és mi fütyülve eldobjuk az elsô 550-et, hogy élvezzük az utolsó ötvenet. Mi nem akarunk ilyen pazarlók lenni, és megbecsüljük a modemezôket is. Ilyenkor jön jól a dinamikus kurzor.

Mivel a dinamikus kurzor eredményhalmaza illékony, másodpercrôl-másodpercre változik, a FETCH ABSOULTE ennél a kurzornál nem támogatott. Hiszen a FETCH ABSOULTE 100 ebben a pillanatban lehet, hogy teljesen más sort választ ki, mint 10 másodperc múlva, ha közben 500 felhasználó püföli a tábla tartalmát. Ennek ellenére *(surprise :)*, a FETCH RELATIVE mûködik. A FETCH FIRST is, így a kettôbôl már össze lehet rakni egy FETCH ABSOLUTE-ot. Hogy miért van ez így? Nem tudom. De mûködik, és ezt nagyon jól ki tudjuk használni.

Az eddigiekbôl az következik, hogy a dinamikus kurzornak kell lenni a leggyorsabbnak, hisz ez nem épít semmilyen átmeneti táblát. Ez egészen addig igaz, amíg csak egy táblából kérdezünk le. Amint illesztéssel *(JOIN)* több táblát összekapcsolunk lehet, hogy gyorsabb az elején felépíteni egy átmeneti táblát az összekapcsolt eredményekbôl, mint mindig menet közben összekapcsolni újra és újra a táblákat. Magyarra fordítva illesztett táblák esetén nem biztos, hogy a dinamikus kurzor a leggyorsabb.

Forward only kurzor

Az elôbbi három kurzor alapértelmezésben támogatja azt, hogy elôre-hátra mozogjunk vele az eredményhalmazban *(SCROLL)*. Azonban az esetek igen jelentôs részében csak elôre akarunk végighaladni a kurzorral, mert egyszerûen kiíratjuk az eredményeket egy riportban vagy egy weblapon. Ebben az esetben adhatunk egy kis könnyítést az SQL Servernek azzal, hogy jelezzük, hogy a kurzorunkkal csak elôre fogunk lépegetni. Ezáltal a kiszolgálónak kevesebb munkába kerül nyilvántartani a pozíciónkat.

A forward only nem egy negyedik kurzortípus, hanem az elôzô három kiegészítése, pontosítása. Pl. egy keyset kurzor egyirányúsítása:

```
DECLARE curTest CURSOR FORWARD_ONLY KEYSET
FOR ...
```

Fast forward only kurzorok

Ez a negyedik típusú kurzorunk. Ez egyetlen hálózati körülfordulásban visszaküldi a teljes eredményhalmazt a kliensre, majd automatikusan zárja a kurzort. Így az ügyfélprogramnak gyakorlatilag csak kérni kell az adatokat, és nem kell foglalkozni a

kurzor megnyitásával - bezárásával. Az ADO dokumentáció hallgat errôl a lehetôségrôl, de ODBC-n keresztül el lehet érni. Mivel azonban manapság már nem nagyon használunk ODBC-t, ezzel a lehetôséggel nem nagyon foglalkozunk. Mellesleg az ADO „sima" forward only kurzora kísértetiesen hasonlít viselkedésben erre a kurzorra. Nem véletlenül. Ha ADO-n keresztül csak olvasható, forward only kurzort kérünk, akkor az egy fast forward only *(régebbi nevén firehouse)* kurzor lesz.

Jogosan kérdezheti bárki, hogy miért foglalkozunk ennyit a kurzorokkal, amikor olyan ritkán használjuk. Biztos? Lehet, hogy a Transact SQL kurzorok felhasználása elég speciális, és csak kevesen fognak belegabalyodni. Azonban mi történik akkor, amikor egy ügyfélprogram ADO vagy ODBC segítségével végrehajt egy lekérdezést az SQL Serveren? Hmmm? Hogyan nyissuk meg a kapcsolatot a szerver felé? Ja, hogy lehet valamit állítani a lekérdezés végrehajtásakor? És még valami kurzortípust is? Ejha, nézzünk csak ennek a körmére!

API kurzorok

Amikor ADO segítségével végrehajtunk egy lekérdezést az SQL Serveren, akkor két választási lehetôségünk van.

1. Teljes egészében letöltôdik az egész eredményhalmaz a kliensre, és ott navigálunk a rekordok között – ekkor beszélünk kliensoldali kurzorról.
2. Jelezzük, hogy kiszolgálóoldali kurzort szeretnénk használni, és akkor csak azok a rekordok kerülnek át az ügyféloldalra, amelyekre ténylegesen rápozícionálunk, és kiolvasunk.

A kliensoldali kurzorok akkor hatékonyak, ha kis *(<1000 sor)* eredményhalmazokkal dolgozunk, és az összes sorral szeretnénk dolgozni. Ilyenkor a teljes eredményhalmaz - kiszolgálóoldali kurzor felhasználása nélkül - egy kötegben átmegy a kliensoldali adatbázismeghajtó programra, ami implementálja az eredményhalmaz navigálásához szükséges függvényeket. A kiszolgáló gyorsan megszabadul a munkától, a zárolások gyorsan feloldódnak. Az ügyfélprogram akármeddig dolgozhat a rekordokon, a szervert nem terheli a ténykedése. És ami még fontos, a navigáció nem generál járulékos hálózati forgalmat.

A kiszolgálóoldali kurzorok akkor használhatók ki jól, ha a lekérdezés eredményhalmaza nagyon nagy, és nem akarjuk feldolgozni az egészet, csak egy részét. A legtöbb böngészô, listázó típusú adat megjelenítés ilyen. Így a kliensprogramot nem árasztja el a kiszolgáló megabájt méretû eredményhalmazzal, megint csak a lassú vonalak végén ülôk örömére, és a bérelt vonali szolgáltatók bánatára.

A kiszolgálóoldali kurzorok esetén ugyanaz a választékunk, mint amint a Transact SQL kurzoroknál már megnéztünk.

Hogyan lehet elképzelni az ADO által megvalósított kurzort? Valahogy úgy, hogy a lekérdezés végrehajtásakor a távolban, a kiszolgálóoldalon létrejön egy kurzor. Nem a DECLARE CURSOR és az OPEN utasításokkal, hanem speciális, csak erre a célra használat „ál" tárolt eljárásokkal, mint az sp_cursor és társai. Ezek a tárolt eljárások a valóságban nem is léteznek *(bár úgy látszanak)*, hanem az SQL Server kernel tudja, hogy egy adatbázismeghajtó program kiszolgálóoldali kurzort akar létrehozni, és ennek megfelelôen belül létrehozza a kurzort. További speciális tárolt eljárások hívásával az ügyféloldali adatbázismeghajtó program FETCH és egyéb kurzormûveletet tud végre hajtatni a létrehozott kurzoron a távolban, a kiszolgálón. Ezeket könnyedén megfigyelhetjük az SQL Profiler felhasználásával.

Ezeket az adatbázismeghajtó program által létrehozott kurzorokat API kurzoroknak nevezzük. Nem úgy, mint a Transact SQL kurzorokkal, ezekkel nap mint nap találkozunk, vagy tudatosan, vagy nem, hisz az ügyfélprogram, adatbázismûveletek használják ôket.

Írjuk meg az altavistát!

Zárásul tegyük fel a koronát a tudásunkra. Tervezzünk egy olyan alkalmazást, ami képes végrehajtani egy paraméterezett lekérdezést, célszerûen olyat, aminek nagyon nagy az eredményhalmaza *(például az altavistán rákeresünk a sex szóra :)*. Azután ebbôl a halmazból jelenítsük meg a 200 és a 220. sor közötti rekordokat.

Egy valódi, például webes alkalmazásban megjelenítenénk egy sorszámozott menüt, amivel a felhasználó közvetlenül kérheti a megfelelô lapok megjelenítését. Valami ilyesmire gondolok:
20 40 60 80 100 120 140 160 180 200 >>400>>

A megoldáshoz az ADO kurzoraihoz folyamodunk. *(Elnézést azoktól, akik szeretik a tiszta SQL kódokat, de most egy kicsit át kell mennünk ügyféloldalra, és VB-re.)*

Mivel a kedvenc Northwind adatbázisomban nincs elég nagy tábla, ezért létre fogunk hozni nagyon nagy teszttáblákat a megfelelô méretû eredményhalmaz reményében. Az Order Details tábla 2155 sorból áll. Ez elég lesz az elsô referenciamérésünkhöz, azonban gyengus lenne demonstrációs célra, hisz azt ígértem a cikk elején, hogy milliós nagyságrendû táblából fogunk kiválogatni sorokat ezredmásodpercek alatt. Lássunk egy teszttáblát, amit a méréseinkhez fogunk felhasználni:

```
CREATE TABLE BigTable1000000(
   nID INT NOT NULL PRIMARY KEY IDENTITY(1,1),
   nCol1 INT NOT NULL,
   nCol2 INT NOT NULL,
   cText1 VARCHAR(500),
   cText2 VARCHAR(500))

--Tartalomgeneráló kód a weben: [1]
```

Nézzük meg a megjelenítést végzô ügyféloldali tesztkódunkat, csak a legfontosabb részleteket tanulmányozva. Deklaráljuk a lapozást definiáló paramétereket!

```
Dim nRecordNumber
Dim nPosition
nPosition = 200     'Ez a kiíratandó első rekord
nRecordNumber = 20 'Ennyi rekodot írunk ki
```

Kapcsolódjunk rá az SQL Serverre!

```
Dim conTest
'Létrehozzunk az ADO Connection objektumot
Set conTest = CreateObject("ADODB.Connection")
'Megadjuk a megfelelő connection string-et
conTest.ConnectionString = "Provider=..."

'kiszolgálóoldali kurzort használunk!
conTest.CursorLocation = adUseServer

'Rákapcsolódunk a kiszolgálóra
conTest.Open
```

Létrehozzuk a nagy eredményhalmazt generáló ADO parancsot.

```
Dim cmdList, rsOrders
'Létrehozunk egy ADO Command objektumot
Set cmdList = CreateObject("ADODB.Command")
'Hozzákapcsoljuk a már felépített Connection-höz
Set cmdList.ActiveConnection = conTest
'Ez az SQL parancs generálja a négymilliós
'eredmény halmazt
cmdList.CommandText = "SELECT ..."
```

Az eredményrekordokat tároló recordset létrehozása:

```
'Az itt létrehozott ADO Recordset objektum
'fogja tárolni a kiszolgálóról érkező
'eredményhalmazt
Set rsOrders = CreateObject("ADODB.Recordset")
'Megjelöljük a rekordok forrását
'(a lekérdezésünket)
Set rsOrders.Source = cmdList
```

Amire kimegy a játék:

```
'Minden jó anyja: a dinamikus kurzor
rsOrders.CursorType = adOpenDynamic

'Az eredményhalmaz (és a kurzor) megnyitása
rsOrders.Open
rsOrders.CacheSize = nRecordNumber
```

Álljunk meg egy pillanatra! Mi az a Recordset.CacheSize? Ezzel állítjuk be a kurzorunk szélességét. A Transact SQL-es példánk kurzora 1 széles volt, azaz minden egyes, a következô rekordra navigáló lépés újabb és újabb adatbázismûvelettel járt. Egy, az SQL Serveren futó programnál ez nem is probléma, de egy ügyfél-kiszolgáló programban az lenne a jó, ha egyszerre több rekord is lejönne a hálózaton, így ritkábban kellene hozzáférni a kiszolgálóhoz további rekordokért. Erre való a CacheSize jellemzô. Ha beállítjuk 10-re, akkor a kurzorunk 10 kövérségû lesz, így az elsô elemre navigáláskor nem csak az elsô rekord töltôdik le, hanem további 9 is. Így a következô 9 elemre történô navigáció nem igényel újabb szerverhez fordulást.
Mozogjunk elôre a 200. rekordra!

```
rsOrders.Move(nPosition)
```

Sok példaprogram igen helytelenül azt sulykolja belénk, hogy használjuk a

```
rsOrders.AbsolutePosition = nPosition 'nem!
```

utasítást a megfelelô rekordra való mozgásra. Azonban mi tudjuk, hogy a dinamikus kurzornál nincs FETCH ABSOULTE, így ez az utasítás is elszáll a

```
ADODB.Recordset (0x800A0CB3)
Current Recordset does not support bookmarks.
This may be a limitation of the provider or of the
selected cursortype.
```

hibával. A hiba második része a mi problémánk. Statikus vagy keyset kurzorral menne az abszolút pozícionálás, csak akkor meg a teljesítmény lesz nagyon siralmas, ahogy azt majd hamarosan látjuk.
Ránavigáltunk a 200. rekordra, most már csak végig kell menni a következô 20-on:

```
Dim nDisplayCount
nDisplayCount = 0
Do While (NOT rsOrders.EOF) And
↳ (nDisplayCount < nRecordNumber)
  Print rsOrders("OrderID")
  nDisplayCount = nDisplayCount + 1
  If nDisplayCount < nRecordNumber Then
  ↳ rsOrders.MoveNext
Loop
```

Addig gyalogolunk elôre a rekordokban, amíg vagy a végére nem érünk, vagy ki nem írattuk a kívánt számú sort.
És természetesen takarítunk magunk után, mert nem szeretjük a memóriát suttyomban fogyasztgató alkalmazásokat:

```
rsOrders.Close
Set rsOrders = Nothing
Set cmdList = Nothing
conTest.Close
Set conTest = Nothing
```

Nem csak a szánk jár, avagy a végsô terhelésteszt

És most jöjjenek az izgalmas teljesítménymutatók, ahol kiderül, hogy megbukik-e az elmélet, vagy megerôsítést nyer *(mivel ez a cikk megjelent, vélhetôleg beigazolódott :)*.
A tesztkörnyezet egy 450 MHz-es PII, 128 Mbyte RAM-mal, Windows 2000 Advanced Serverrel, az ügyfél és a szerver egy gépen volt. Az ügyfélalkalmazás egy ASP-ben megvalósított VBScript kód, amely a **[1]** címen élôben ki is próbálható, valamint a teljes forráskód is letölthetô. Az alkalmazás a fentebb ismertetett példa kibôvített változata, de a lényege azonos azzal.
Az SQL Server végrehajtási idôket SQL Server Profiler-rel mértem, összeadva a parancs végrehajtásához szükséges összes mûvelet által használt idôket. Az ügyfél végrehajtási idôt az ASP kód kezdete és vége közt mértem, ebben benne van a kapcsolat kiépítése, a parancs végrehajtása, a rekordokon való végig gyaloglás, és a kapcsolat lezárása is.
Az elsô lekérdezést az Order Details tábla ellenében hajtottam végre.

```
SELECT OrderID, Quantity FROM [Order Details]
```

Ez a tábla még elég kicsi ahhoz (2155 sor), hogy az összes kurzort kipróbálhassuk vele - véges türelemmel is.
A tesztben a leválogatott adatokban elmozogtam a 200. rekordig, majd onnan a következô húszat írattam ki.
Lássuk az így mért eredményeket:

Kurzor	SQL Server végrehajtási idô [sec]	Ügyfél végrehajtási idô [sec]	Logikai diszk mûvelet [lap]
Dinamikus	<0,001	0,03	106 olvasás 2 írás
Forward only	0,01	0,02	10 olvasás 0 írás
Keyset	0,110	0,13	4521 olvasás 11 írás
Statikus	0,05	0,07	4374 olvasás 7 írás

Mit mondanak az eredmények? A dinamikus kurzort a szerver nagyon gyorsan végrehajtja, gyakorlatilag a végrehajtási idôre az SQL Server Profiler 0-t ad vissza. Ennek ellenére kliensoldalról szemlélve a forward only kurzor teljesített legjobban, és a logikai olvasásokban is az vezet. Nem véletlenül van az, hogy kis táblák tartalmának egyszerû kilistázására javasolják a forward only kurzort. Nagyon kicsi szerveroldali terhelést okoz, és a kliens nagyon gyorsan megkapja az eredményhalmazt. A profiler-ben látszik, hogy valójában ilyenkor az OLE-DB szolgáltató nem is nyit meg kurzort a szerveren, csak az alapértelmezett eredményhalmazt tölti le. Valamit azért elmond a szervernek a lekérendô eredményhalmaz hosszáról, ami azonban nem látszik a profiler-ben. Viszont az látszik, hogy minél több rekorddal léptetjük elôre a *(nem is létezô)* kurzort, annál több lapot olvas be.

A táblázatot szemlélve egy további furcsa eredmény üti meg a szemünket. A keyset kurzor lassabb, mint a statikus kurzor! Ez ellentétben áll azzal, amit a mûködése alapján várnánk tôle. Egy tippem van, miért van ez. A táblánk kicsi *(~2000 sor)*. Ennyit a szerver pillanatok alatt átmásol a tempdb-be, ahonnan a kliensnek már nagyon gyorsan, közvetlenül kiszolgálja a kívánt sorokat. A keyset kurzornál a kulcsokat még gyorsabban be tudja másolni a tempdb-be, mint az elôbbi esetben, azonban kiolvasáskor minden egyes kulcshoz elô kell bányászni az adatokat az eredeti táblából. Ilyen kisméretû táblánál nagyobb a bányászás *(bookmark lookup)* költsége, mint átmásolni az összes eredményt egy másik táblába. Ez nem látszik triviális módon a lekérdezésbôl.

Töltsük fel a már emlegetett teszt táblánkat 10000 sorral, és válogassuk le az egészet (a tesztadatokat generáló script – terjedelmi okokból – a [1] címen érhetô el):

Kurzor	SQL Server végrehajtási idô [sec]	Ügyfél végrehajtási idô [sec]	Logikai diszk mûvelet [lap]
Dinamikus	<0,001	0,02	105 olvasás 0 írás
Forward only	0,15	0,15	5 olvasás 0 írás
Keyset	0,22	0,22	20422 olvasás 0 írás
Statikus	0,25	0,26	20552 olvasás 0 írás

A dinamikus kurzor fittyet hány a tábla méret változtatásra, ô ugyanolyan gyors maradt. A forward only kurzor jelentôsen lassult, azonban emberi léptékkel nézve még mindig villámgyors. A másik két kurzor már kezd belefulladni az adatokba. A lapolvasások száma a tábla sorainak számával arányban nôtt. Ennek van egy nagyon fontos tanulsága. Ha elkészítünk egy alkalmazást, amiben nem ügyelünk a kurzor típusára, és az statikus vagy keyset lesz, akkor eleinte jó gyorsnak fog tûnni a program, mert az összes sort képes benntartani a szerver a fizikai memóriában, és ott a mai processzorok nagyon gyorsan tudnak keresni. Ha azonban az alkalmazás éles üzeme során az adatok elkezdenek záporozni a táblákba, akkor lehet, hogy egy hónap múlva az alkalmazás kifekszik, jönnek az idôtúllépésrôl szóló üzenetek, és a haragos ügyfelek.

A helyzet akkor lesz csak igazán drámai, ha akkorára dagadnak a táblák, amekkorát már soha nem tud egyben benntartani a szerver a memóriában, így elindul a merevlemez reszelés. Ekkor mutatja csak ki a foga fehérjét a rossz tervezés. Nézzük csak meg egy 1,000,000 soros táblával az elôbbi számok változását:

Kurzor	SQL Server végrehajtási idô [sec]	Ügyfél végrehajtási idô [sec]	Logikai diszk mûvelet [lap]
Dinamikus	<0,001	0,03	108 olvasás 0 írás
Forward only	0,17	0,18	17 olvasás 0 írás
Keyset	83,11	83,26	2,845,123 olv. 3276 írás
Statikus	281,466	288,68	2,988,655 olv. 8432 írás

Gyakorlatilag a két utolsó kurzor kiesett a játékból, az ADO alapértelmezett 30 másodperces parancs idejébôl már rég kifutottak volna a parancsaink. És most csak egy felhasználó terhelte a szervert, nem 1000! Egy élô rendszer ebben az esetben egyszerûen leállna. Sajnos ilyen tervezési hibák miatt elég gyakori, hogy alaptalanul szidnak egy rendszert, merthogy az már egymillió sort sem tud rendesen kezelni. Bezzeg a pityipalkó cég terméke...

Akkor tessék csak megnézni azt az elsô sort! Végrehatási idô a kliens oldalon mérve is 30 milliszekundum! Pedig a tábla már egymillió sort tartalmaz, ami messze meghaladja az én gépem RAM-jának tárolókapacitását. De nem is ez a lényeg. Hiába nôtt a tábla mérete az elsô esethez képest az 500 szorosára, a végrehajtási idô gyakorlatilag nem változott. Ezért szeretem én úgy a dinamikus kurzort. Szeressék Önök is, és éljenek vele!

Zárszó

A [1] címen nem csak példakódok találhatók, hanem minden, a cikk második felében szereplô kódot interaktívan kipróbálhat a kedves Olvasó, maga választva ki a mérés összes paraméterét, a kurzortípusokat stb.

A kurzorok elméletének egyik sarkalatos pontja, a zárolás teljesen kimaradt a mostani cikkbôl, a szokásos terjedelmi okok miatt. Azonban a jövô hónapban egy teljes cikket szánok a zárolások lelki világának megértésére, mert - a kurzorokhoz hasonlóan - a nem megfelelô alkalmazásuk szintén ledegradálhatja az alkalmazásunk teljesítményét - és vele együtt minket is.

Jó kurzorizálást kívánok a web-en és a földön egyaránt!

Soczó Zsolt MCSE, MCSD, MCDBA
Netacademia Kft.

A cikkben szereplô URL-ek:
[1] http://technet.netacademia.net/feladatok/sql/cursor

Transact SQL *(VI. rész)*

Bevezetés

A mai, korszerû adatbázisok egyik legfontosabb jellemzôje, hogy sokan használják egyidejûleg. Mivel a felhasználók, alkalmazások egymástól függetlenül próbálják meg módosítani a táblák tartalmát, gyakori a konfliktushelyzet. Ilyenkor kezdenek lelassulni a rosszul megtervezett adatbázisok, és jönnek az idôtúllépésrôl, valamint a misztikus dead-lock-okról szóló hibaüzenetek, nem beszélve a logikailag hibás adatokról. Ebben a részben részletesen kitárgyaljuk a zárolások okait és fajtáit, a következô számunkban pedig a dead-lock-ok misztikus világáról lebbentjük le a fátylat. Cikksorozatunk mostani fejezete elég nehéz, ám annál fontosabb témakörrel foglalkozik, ami nélkül igen nehéz megbízható és hatékony adatbázisokat tervezni az SQL Server 2000-re.

Optimista vagy pesszimista?

Nézzünk meg egy klasszikus ügyfél-kiszolgáló alkalmazást, ami kurzorok használatával módosítja az adatokat. A legtöbb Visual Basic és Visual C++ alkalmazás ilyen.

Tegyük fel, hogy van egy adatbázis, amely egy cég alkalmazottait tartja nyilván. Kiss Béla cégen belüli pozíciója megváltozik, kap egy Senior jelzôt a rangja elé. Ezzel együtt a lakcíme is megváltozott, amirôl külön értesíti az egyik HR-es hölgyet. Az emberi erôforrás menedzsmenten lelkes emberek dolgoznak, és azonnal nekilátnak a változás adatbázisba rögzítésének. A lelkesedésük nagyobb, mint a munkaszervezettségük, így egyszerre ketten kezdik el módosítani a kérdéses alkalmazott adatait az adatbázisban. Tegyük fel, hogy mindkettejük elôtt ki vannak listázva Béla adatai, és nekiállnak módosítani a rekordot. Az elsô a lakcímet és a rangot, a második csak a rangot írja át. Megnyomják az „Elment” gombot, és tegyük fel, hogy az elsô hölgy a gyorsabb. Mi történik?

Két eset lehetséges. Egy butább adatbáziskezelô vagy egy rosszul megírt ügyfélalkalmazás esetén az elsô ügyfél által kért változtatás beíródik az adatbázisba, amit követ a második ügyfél módosítása. Mivel mindketten ugyanazokból a kiinduló adatokból módosított adatokat írják vissza, a második módosítás fejbe csapja az elsôt, azaz a végleges rekordban nem lesz módosítva a lakcím, mert a második hölgy csak a rang mezôt módosította. A probléma nem az, hogy ez így megtörténhet, hanem az, hogy az ügyfélprogramok nem is szereznek róla tudomást, hogy módosításvesztés történt.

Az iménti helyzetben felvázolt helyzetet hívjuk az elveszett módosítások problémájának. Hogyan védekezik az ilyen helyzetek ellen egy okos adatbáziskezelô? Amikor egy ügyfélprogram lekér egy adott rekordot az adatbáziskezelôtôl, akkor a szerver megjegyzi, hogy valaki letöltötte a rekordot, mert módosítani szeretné. Amikor más ügyfelek is szeretnék ugyanezt megtenni, akkor kétféle dolog történhet. Ha az elsô ügyfél optimista zárolás felhasználásával kérte le a rekordot, akkor a hasonlóan eljáró további ügyfelek is megkapják a rekordot. Azonban módosítás visszaírási kísérlet esetén az adatbáziskezelô megnézi, hogy megváltozott-e az adatbázisban tárolt sor a korábban lekért állapothoz képest *(annyira nem optimista, hogy vakon megbízzon benne, hogy nem változott* :). Ha igen, akkor a próbálkozónak már csak egy hibaüzenet jár, ami arról tájékoztatja, hogy a módosítani kívánt rekordot már valaki más módosította:

```
Optimistic concurrency check failed. The row was
modified outside of this cursor.
```

Ilyenkor nincs mit tenni, újra le kell kérni a módosított adatokat, újra beírni a változtatásokat, és újra megpróbálni beküldeni a változtatási kérelmet. Ha ezúttal mi voltunk a leggyorsabbak, akkor nyertünk, és a mi módosításunk lesz érvényes. Ha nem, try again... Nyilvánvaló, hogy egy olyan rendszerben, ahol gyakoriak a módosítások, ott nem megfelelô ez az eljárás, mert túl gyakoriak az ütközések.

A másik stratégia úgy gondolkodik, hogy ne ringassuk hiú ábrándokba a második, harmadik, satöbbi ügyfelet, hanem az elsô alkalmazás, ami módosítani akar egy rekordot lefoglalja azt, és a többiek addig nem is tudják lekérni a rekordot mindaddig, amíg az elsô fel nem oldja a zárolást. Ezt a stratégiát pesszimista zárolásnak hívjuk. Ez is egy elfogadható hozzáállás, ráadásul egyszerûbb implementálni a várakozást, mint lekezelni a sikertelen módosítást. *(Gyakorlatilag nem kell tenni semmit, mert az adatbázist elérô metódus nem tér vissza addig, amíg a módosítandó rekord fel nem szabadul.)* Például egy helyfoglaló rendszernél csak ez a módszer tud helyesen mûködni, hisz optimista esetben az operátor még szabadnak láthat olyan helyeket, amelyeket már rég lefoglaltak más operátorok. Inkább ne is láthassa azokat a helyeket, amelyeket éppen valaki más próbál lefoglalni.

Az eddigi példában olyan helyzetrôl beszéltem, amikor a rekordokat kurzor segítségével kértük le, és a kapott recordseten keresztül módosítjuk az adatokat. Ez a fajta megoldás a mai világban egyre ritkább, és különösen a Webalkalmazásokban nem ilyen módon kezeljük az adatokat. Azokban általában tárolt eljárások segítségével módosítjuk a sorokat. Ilyenkor már nagyon könnyen fejbe lehet csapni a konkurens módosítások eredményét, hisz az adatok lekérése és a módosított adatok visszaírása közben megszakad az ügyfélprogram *(a Webalkalmazás)* kapcsolata az adatbázissal, így az adatbázisnak még esélye sincs arra, hogy zárolással vagy Voodoo varázslással megóvjon minket a módosítások elvesztésétôl.

Tegye a szívére a kezét minden Webalkalmazás fejlesztô! Gondolt már valaha erre a problémára? Vagy csak mechanikusan visszaírja a módosított eredményeket a forrástáblába? Vesszen a lassabb? Az ADO természetesen ilyen helyzetekre is biztosít megoldást, de használjuk ezeket? *(A jövôben mindenképpen áldozunk egy-két cikket a témának.)*

SQL Server tranzakciók

Szakadjuk el egy kicsit a kurzort használó ügyfélprogramoktól, és evezzünk át a tiszta SQL Server megoldásokhoz, valamint a tárolt eljárásokat használó alkalmazásokhoz. Nézzük meg, hogy a tranzakciók során mennyire vagyunk védettek mások adatmódosításai ellen.

Kiindulásként álljon itt egy kérdés. Alapértelmezett beállítások mellett biztos lehetek benne, hogy egy tranzakción belül háborítatlanok maradnak az általam használt táblák, miközben mások is dolgoznak az adatbázisban? Legtöbben azt gondolják, igen. Ha biztos akarok lenni abban, hogy a tábláimat nem változtatják meg a hátam mögött a tranzakcióm alatt, akkor elég BEGIN TRAN és COMMIT TRAN közé rakni az utasításaimat, és minden rendben lesz? Biztos? Egyáltalán nem. Járjuk körbe ezt a témát, mert ennek megértése nélkül senki nem mondhatja el magáról, hogy konzisztens adatbázist tud tervezni.

A zárolások fajtái

Annak érdekében, hogy az SQL Server szabályozni tudja az adatokhoz való párhuzamos hozzáféréseket, a védendô adatokra zárolásokat helyez el. Az SQL Serverben többféle zárolási típus van, és mindegyiknek van egy meghatározott viselkedése. Például más zárolást kell használjon a szerver az adatok olvasása során *(SELECT)*, hisz ilyenkor általában csak azt kell megakadályozni, hogy más tranzakció módosítsa az éppen olvasás alatt álló adatokat. Ezzel szemben például egy adatmódosító tranzakció közepette nem lenne szerencsés engedni a többi tranzakciót, hogy olvassa az éppen módosítás alatt álló adatokat, plane, hogy módosítsa ugyanazt. Nyilván ehhez másféle zárolásra van szükség. Tekintsük át a legfontosabb zárolási típusokat!

Mint említettük az adatok olvasása során meg kell akadályozni, hogy az éppen kiolvasott adatokat mások módosítsák az olvasási mûvelet közben, de meg kell engedni, hogy mások is olvashassák, hisz az veszélytelen a mi tranzakciónkra nézve. Ehhez az SQL Server Shared lock-okat helyez el az olvasott adatokra *(a könnyebb követhetôség kedvéért nem fordítottam le a zárolások nevét, és az egyszerûbb olvashatóság miatt a zárolás eredetijét, a lock-ot is meghagyom)*.

Ha egyszerre több tranzakció is olvassa ugyanazt az adatot, akkor mindegyik elhelyezi a maga Shared lock-ját a rekordokon, és addig rajta is tartja, amíg nem végez az olvasással. Az adatmódosító utasítások *(INSERT, DELETE és UPDATE)* alatt nem szabad másnak olvasni a módosítandó adatokat, ilyenkor a szerver Exclusive lock-ot helyez el a sorokon. Az Exclusive lock mellett más nem helyezhet el semmilyen zárolást a sorokra, meg kell várnia, míg az adatmódosítás befejezôdik, és a tranzakciót így vagy úgy, de be nem fejezik. A legtöbb esetben ezzel az esettel kerülnek szembe az adatbázisfejlesztôk és üzemeltetôk, azaz, hogy egy hosszú ideig tartó adatmódosító tranzakció zárol egy bizonyos adatmennyiséget, így az egyéb adatolvasó vagy módosító tranzakcióknak várni kell a módosítás befejezéséig. Ezt sokan tévesen deadlock-nak azonosítják, pedig ennek semmi köze nincs ahhoz. Egyszerûen csak egy hosszú idejû tranzakció blokkolja a többi tranzakció munkáját. Az SQL Server Enterprise Manager Management, Current Activity, Lock/Process ID alatt találhatjuk meg a szerveren a zárolásokat megjelenítô grafikus alkalmazást. Ennek segítségével azonosítható az a tranzakció, ami blokkolja a többit *(felkiáltójeles emberke ikon)*. Ezen a nyomon elindulva meg lehet keresni, és át lehet írni a bûnös tranzakciót. Aki nem szereti a grafikus felületeket, annak az sp_lock tárolt eljárást ajánlom a zárolások megfigyelésére.

Az UPDATE rendhagyó mûvelet a többi háromhoz képest, mert az elsô fázisban fel kell olvasnia a módosítandó adatokat, a másodikban pedig módosítani azt. Emiatt az olvasási részben Shared lock-ot kell elhelyezzen az adatokon, a módosítás során pedig Exclusive lock-ot. Az ô kettôs természete miatt kapott is egy saját zárolási típust, amit Update lock-nak hívnak. Az UPDATE az adat olvasási fázisban Update lock-ot rak a sorokra, és a tényleges módosítás megkezdés elôtt felemeli azt Exclusive lock-ra. Azért nem Shared lock-ot használ, mert az Update lock nem engedi meg, hogy mások is igényeljenek Update lock-ot ugyanazokra az adatokra, így nem tudja más megmódosítani az adatokat a felolvasás és a módosítás között. A dead-lock-ok megelôzésében nagyon fontos szerepe van az Update lock-nak, amirôl a következô számban írok bôvebben.

Schema Modification lock-ot az adatbázis szerkezetét módosító utasítások *(például ALTER TABLE)* helyeznek el a megfelelô objektumokon, hogy közben nehogy mások is megpróbálják ugyanazt módosítani.

A lekérdezések fordítása közben a szerver Schema Stability lock-al akadályozza meg a lekérdezésben szereplô táblák és egyéb objektumok szerkezetének módosítását.

A zárolások finomsága

Eddig elég homályosan fogalmaztam meg, hogy az SQL Server valójában mekkora adatmennyiségeket zárol a tranzakciók során. Most nézzük meg, hogy milyen egységekben tud adatokat zárolni a szerver.

A legfinomabb zárolási egység a sor. Ez képes egyetlen rekord zárolására, azaz miközben egy sort módosítunk, egy másik tranzakció képes a mellette található sor *(reklord)* olvasására vagy módosítására.

Ha egy lapon *(8 kByte-os egység, amely a sorokat tartalmazza)* sok sort kellene zárolni, akkor a szerver inkább zárolja a teljes lapot, ahelyett, hogy sok sor-zárolást kellene nyilvántartania. Egyes esetekben, amikor olyan sok módosítás történik, hogy az szinte egy egész tábla tartalmát érinti, a szerver inkább zárolja az egész táblát, semmint egyedi lapokat, ezzel a zárolások nyilvántartásához szükséges erôforrásokat spórolva.

Az SQL Server automatikusan választja ki, hogy mikor milyen finomságú zárolásra van szükség. A tranzakció által érintett sorok számától függôen keres olyan szintû zárolást, ami még elég finom ahhoz, hogy ne korlátozza jelentôsen a többi tranzakció futását, de ne is kelljen nagyon sok lock-ot nyilvántartania. A szerver egy tranzakció lefutása közben is képes változtatni a zárolás finomságát. Lehet, hogy elindul sorzárolással, ám a sorok zárolása közben észreveszi, hogy már olyan sok sort kell nyilvántartania, hogy érdemesebb lenne áttérnie az egész tábla zárolására. Ezt a folyamatot, amikor egy finomabb, de nagy számú zárolásról a szerver áttér egy durvább, nagyobb tartományra ható, de kevesebb számosságú zárolásra zárolás eszkalációnak *(Lock Escalation)* hívjuk. Ha tudjuk, hogy a tranzakciónk nagyon sok sort fog érinteni, akkor lehet, hogy érdemes a szervernek súgni, hogy nem érdemes sorzárolástól indulva végiglépkednie a zárolásokon, hanem rögtön kezdje például tábla szintû zárolással. Lehet, hogy így olyan tranzakciókat is blokkolunk, amelyeket sor vagy lap zárolással nem befolyásolnánk, de a kis számú zárolás nyilvántartása miatt a tranzakciónk lehet, hogy sok-

kal gyorsabban fut le, így végeredményben kevesebb blokkolást okozunk a többi tranzakció felé.
Más esetben lehet, hogy az SQL Server egy egész táblát zárolna, és így más tranzakciók nem tudnának abban dolgozni, például adatokat beszúrni. Tipikus példa erre, amikor egy hosszú idejû lekérdezést futtatunk, ami múltbeli adatokkal foglalkozik, miközben záporoznak be a táblába a mai naphoz tartozó sorok. Lehet, hogy a lekérdezés akár a sorok elsô 99%-át érinti, így a szerver nyilvánvalóan egy darab tábla zárolással lefoglalja a tranzakciónk számára a táblát, ám így az adatokat beszúró alkalmazás nem tud írni sorokat a tábla végébe. Ilyenkor lehet, hogy például lapszintû zárolást erôltetve a tranzakciónk nem 5 perc, hanem fél óra alatt fut le a sok zárolás adminisztrációja miatt, de eközben az adatokat beszúró alkalmazás egy pillanatig sem állt le. Azaz vannak esetek, amikor szélesíteni akarjuk a zárolások tartományát, és vannak, amikor szûkíteni, az alkalmazásunk logikájától függôen. Hogyan befolyásolhatjuk az SQL Servert a zárolások finomságát illetôen? A kérdésre a lock hint-ek adnak választ, a cikk utolsó részében.
A végére hagytam egy különleges zárolási típust, amely az elôbbiekkel ellentétben nem fix méretû zárolást valósít meg. Ez az index-tartományzárolás. Bizonyos esetekben *(SERIALIZABLE tranzakciók, lásd késôbb)* szükség van arra, hogy egy lekérdezés WHERE feltételében definiált határok között ne lehessen új adatokat beszúrni. Például lekérdezzük az 5 és a 13 közötti azonosítójú sorokat, és nem szeretnénk, ha a tranzakciónk alatt valaki más beszúrna új sorokat olyan azonosítóval, amely 5 és 13 közé esik. Ebben az esetben a szerver az indextartomány két végét lezárja Key lock-kal, így a megadott tartományba nem enged új sorokat beszúrni. Ennek a zárolásnak a hossza nyilvánvalóan nem fix, hanem a lekérdezés függvénye. Természetesen ez a zárolás csak akkor tud mûködni, ha a tartományokat definiáló mezôre van index létrehozva. Ha nincs, akkor a szervernek nincs mit tennie, tábla zárolást kell alkalmaznia.

Zárolás kompatibilitás

Mi történik, ha az egyik tranzakció zárolásokat helyez el bizonyos adatmennyiségen, miközben mások ugyanezt akarják megtenni, ugyanazokra az adatokra? Ez attól függ, hogy milyen zárolás van éppen az adatokon, és milyet igényel egy másik tranzakció.
Vannak zárolások, amelyek szeretik egymást, és vannak, amelyek nem. Nyilvánvaló, hogy a Shared lock szereti a Shared lock-ot, azaz, ha az egyik tranzakció olvassa az adatokat, és emiatt Shared lock-okat helyez el az olvasott sorokon, a másik tranzakció veszélytelenül felolvashatja ugyanazokat a sorokat, azaz ô is elhelyezheti a Shared lock-jait ugyanazokon a sorokon. Ha eközben egy harmadik résztvevô is beszáll, aki módosítani akarja a kétszeresen is zárolt *(Shared módon)* sorokat, akkor neki bizony várnia kell egészen addig, amíg a másik két tranzakció be nem fejezi az adatok olvasását, és le nem veszi a lock-jait. Ez is a klasszikus blokkolás esete, amikor egy adatmódosító utasításnak várnia kell arra, hogy elhelyezhesse az Exclusive lock-jait az adatokon. Miután kivárta a sorát, és felrakta a kizárólagosságát biztosító zárolását, senki más semmilyen zárolást nem tud elhelyezni mindaddig, amíg az be nem fejezi a módosító tranzakciót, és le nem veszi az Exclusive lock-ot. Nyilván ebbôl adódik e zárolás neve is.
Azaz abban az esetben, ha egy tranzakció szeretne valamilyen zárolást elhelyezni egy adathalmazon, az SQL Server ellenôrzi, hogy a már fennálló zárolások alapján kiadható-e a kért típusú zárolás. Ha igen, akkor megkapja, a zárolás feljegyzésre kerül, és a trónkövetelô tranzakció megkezdheti a munkáját. Amennyiben viszont olyan zárolást kért, ami logikailag nem összeegyeztethetô a már meglévôkkel, akkor a zárolást kérô utasítást a szerver mindaddig felfüggeszti, amíg meg nem szûnnek az akadályozó zárolások. Az igényt természetesen feljegyzi, és a többi zárolás fokozatos „lehullása” alatt mindig ránéz, hátha már kiadható a kért zárolás. Miközben az igénylô vár a lock-jára, lehet, hogy más tranzakciók is jelentkeznek zárolási igénnyel, és azok között akár olyan is lehet, ami összeegyeztethetô lenne a már fennálló zárolásokkal. Ilyenkor mit tegyen a szerver? Engedje ôket, hogy elhelyezzék a saját zárolásaikat, vagy addig ne engedje ôket szóhoz jutni, amíg a már régóta várakozó tranzakció meg nem kapja az áhított zárolását? Ha engedi ôket, akkor azok lefuthatnak a várakozó elôtt, ám elôfordulhatna az, hogy a sok újabb és újabb kérô soha nem engedné, hogy a várakozó megkapja a zárolását. Azaz ezzel a stratégiával kiéheztetnénk azokat a tranzakciókat, amelyek olyan zárolásokat kérnek, amelyek általában nem kompatíbilisak a már meglévôkkel. A gyakorlatban ez azt jelentené, hogy egy módosító utasítás soha nem kapná meg az Exclusive lock-ját, ha az egymás után érkezô olvasó *(SELECT)*, Shared lock-okat elhelyezô utasításokkal operáló tranzakciók idôben átlapolják egymást. Nyilván ezt nem engedhetjük meg. Emiatt az SQL Server nem engedi zárolni a további kérôket, amíg a már fennálló zárolási igényeket nem elégítette ki. Ez persze azt is jelenti, hogy egy adatmódosító utasítás után akár hosszú sorokban állhatnak a csak olvasni akaró tranzakciók, akik ugyan nyugodtan olvashatnák a Shared lock-kal védett sorokat, de nem tehetik, mert a módosító utasítás vár az Exclusive lock-jára. Gyönyörû hosszú blokkolási láncok tudnak így kialakulni. Mit lehet tenni ellenük?
A legegyszerûbb védekezés, hogy a módosító tranzakciókat nagyon rövidre tervezzük. Nem szabad egy adatmódosító tranzakcióba felhasználói beavatkozásra váró rutint elhelyezni! Mi van, ha közben elmegy ebédelni? Mire visszaér, az adatbázis adminisztrátor már a tízezredik feltorlódott tranzakciót fogja látni a le nem zárt módosító tranzakció miatt! Természetesen ezt nem szabad megengedni.
A másik eszközünk a zárolás finomságának állítása, azaz nem hagyjuk, hogy a módosító tranzakció túl nagy falatot zároljon le kizárólagosan a táblákból. Erre valók a lock hint-ek, amelyekrôl hamarosan szólok.
Egy valamirôl még nem beszéltem. Honnan tudja az SQL Server, hogy melyik zárolási típus melyik másikkal kompatíbilis? Nos, erre a célra van egy táblázata, és abból olvassa ki. Ezt a táblázatot az SQL Server tervezôi alkották meg, figyelembe véve az egyes zárolások természetét, és hogy melyik futhat párhuzamosan a másikkal anélkül, hogy az adatbázis épségét veszélyeztetné. A Books Online a Lock Compatibility címû fejezetben ismerteti ezt a táblázatot.

Az Intent lock-ok

Megnéztük, hogy az SQL Server csak akkor helyez el egy újabb zárolást ugyanazon az adaton, ha az igényelt zárolási típus kompatíbilis a már fennállóval. Azonban hogyan hasonlít össze különbözô finomságú zárolásokat? Ha van egy Exclusive lock egy soron, akkor rakható Shared lock ugyanarra a táblára? Ilyen kérdôjeles helyzet nagyon sok kialakul, hiszen minden tranzakció más finomságú zárolást használhat. Nézzünk erre egy példát.

Az elsô tranzakció Shared lock-kal lefoglal 3356 sort egy táblában. Egy másik tranzakció lefoglal 10 lapot Exclusive módon. Van még 23 éppen futó tranzakció, amelyek 12354 darab Shared és 5 darab Exclusive lap szintû lock-ot tartanak a táblán. Ezután egy sokadik tranzakció tábla szinten szeretne Shared lock-ot. Mit tud tenni a szerver, hogy megállapítsa, megkaphatja-e? Végig kell néznie az összes *(3356+10+12354+5 darab)* zárolást, és meg kell keresnie, hogy van-e köztük olyan, amelyik Exclusive módon birtokolja a tábla valamely szeletét. Ha van, akkor nem adhatja ki a tábla szintû Shared lock-ot. Ha közben egy-egy tranzakció befejezôdik, és engedi el a zárolásait, akkor a lock manager-nek minden esetben végig kellene nézni az összes még megmaradt zárolást, hogy maradt-e még Exclusive, és ha már nem, akkor kiadható a tábla szintû Exclusive lock. Ez az eljárás igen lassú volna. Ennek elkerülésére az SQL Server trükkösen foglalja le a kisebb finomságú *(sor, lap, extent)* zárolásokat. Ha egy tranzakció elhelyez akár csak egy sornyi zárolást is egy táblán, akkor ezzel együtt a szerver elhelyez egy ugyanolyan típusú *(Shared vagy Exclusive)* lock-ot a sort tartalmazó lapra és táblára is, ám azt csak szándéknyilatkozatként Intent Shared vagy Intent Exclusive-ként megjelölve. Ezek után a teljes táblára Exclusive lock-ot kérô tranzakció igénye könnyen eldönthetô, hisz elég megnézni, hogy van-e nem kompatíbilis Intent lock a táblán.

Ez az eljárás nemcsak tábla szinten mûködik, hanem minden olyan szinten, amikor egy kisebb finomságú zárolást kér egy tranzakció. Így egy Exclusive sor lock-ot kérô tranzakció kap egy „valódi” Exclusive lock-ot a soron, és kap egy lap és tábla szintû Intent Exclusive-et is. Ha az Intent lock-ok elhelyezése közben kiderül, hogy a sort tartalmazó lapon már van egy Shared lock, akkor a sorra sem szabad kiadni az Exclusive lock-ot, mert elôfordulhat, hogy belemódosítunk olyan sorba, amit valaki más olvas lap szinten *(pont ezért rakott rá Shared lock-ot)*. Azaz az exkluzív sor-zárolás kiadását megakadályozhatja egy, a sort tartalmazó lapon már létezô Shared lock, így az Intent lock-ok elhelyezése *(helyesebben meghatározása)* közben kiderül a zárolási igény kompatibilitási kérdése is.

A tranzakciók elszigeteltségi szintjei

Láttuk, hogy a párhuzamosan futó tranzakciók többé-kevésbé hatnak egymásra, befolyásolják egymás mûködését. Természetesen egy adatbázisban nem alapozhatunk „többé-kevésbé” szabályokra, valamilyen egzakt módszer kell annak eldöntésére, hogy miközben az egyik tranzakció valamit mûködik egy táblán, a többi tranzakció ebbôl mit lát, illetve mit tehet a kérdéses táblával. Ennek a kérdésnek a szabályozásával az ANSI SQL 92-es szabvány részletesen foglalkozik, és ad is ajánlást egy lehetséges megvalósításra. A szabvány a tranzakciók elszigeteltségét négy szintre bontja. Minél inkább haladunk elôre a szintekkel, annál kevesebb hatással vannak egymásra a tranzakciók, cserébe annál kisebb az esély a tranzakciók párhuzamos végrehajtására. Az egyik oldalon nyerünk valamit, cserébe a másikon veszítünk. SQL Serverben az elszigeteltségi szinteket a tranzakciók belsejében lehet beállítani a

```
SET TRANSACTION ISOLATION LEVEL szint
```

utasítással. Az utasítás hatására a tranzakcióban szereplô összes SELECT utasítás az adott elszigeteltségi szintnek megfelelôen fogja olvasni az adatokat, illetve elhelyezni a zárolásokat a már olvasott adatokon. A tranzakció belsejében bármikor át lehet térni más elszigeteltségi szintre, és onnantól kezdve a SELECT-ek annak megfelelôen fognak mûködni. Ez azonban nem jelenti azt, hogy az elôtte levô SELECT-ek által lefoglalt zárolások feloldódnának, csak azt, hogy az ezután kiadottak az új szintnek megfelelôen fognak viselkedni. Igazából nem sok szituáció indokolja a szintek váltogatását egy tranzakció során, általában az elején beállítunk egy nekünk megfelelô szintet, és azt használjuk a tranzakció végéig. Lássuk hát a négy szintet!

1. READ UNCOMMITTED *(dirty read)*

Ezen a szinten a tranzakcióban szereplô utasítások bármilyen adatot kiolvashatnak a táblákból, függetlenül attól, hogy az adott sort/lapot/táblát zárolta-e valamely más folyamat. Ez azt is jelenti, hogy olyan adatokat is olvashat, amik még nincsenek véglegesen lerögzítve az adatbázisba, azaz a módosító tranzakció végén még nem volt COMMIT TRAN, és lehet, hogy a következô pillanatban visszavonják. Másképpen fogalmazva fizikailag helyes adatokat fogunk kiolvasni, azonban logikailag nem biztos, hogy helyeset.

Ez az elszigeteltségi szint üzleti tranzakciókban elfogadhatatlan, hisz ott csak akkor fogadhatunk el egy adatot érvényesnek, ha az ôt beszúró vagy módosító folyamat véglegesítette a változtatását.

Azonban sokszor nem fontos az adatok hajszálra menô precizitása, de fontos, hogy a tranzakciónk ne blokkoljon más tranzakciókat a sok és hosszú idejû kiolvasás által generált zárolásokkal, valamint, hogy a módosító tranzakciók ne akadályozzák a lekérdezésünk futását. Általában statisztikák és trendek analízise, kimutatások és összesített eredmények számolása során nem baj, ha beveszünk a számításba néhány olyan sort, amelyek esetleg egy másodperc múlva már nem is léteznek, de cserébe gyorsan lefut a tranzakciónk. Ilyenkor nagyon jól jön ez az elszigeteltségi szint.

Nézzük meg, hogy ezen a szinten egy SELECT hatására milyen zárolások keletkeznek az adatbázisban:

```
BEGIN TRAN
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT
  *
FROM
  Employees
WHERE
  LastName LIKE 'B%'
EXEC sp_lock @@SPID

COMMIT TRAN
```

Kimenet:

```
LastName             FirstName
-------------------- ----------
Buchanan             Steven

(1 row(s) affected)

spid dbid ObjId    IndId Type Resource Mode Status
---- ---- -----    ----- ---- -------- ---- ------
51   6    0         0    DB            S    GRANT
```

A kimenetben csak azokat a sorokat hagytam meg, amelyek a 6-os dbid-jû adatbázisra vonatkoznak, ami a vizsgált Northwind. Mit jelent ez a kimenet? Az elsô zárolásokról szóló sor azt mutatja, hogy szerver elhelyezett egy Shared lock-ot a 6-os adatbázisra *(Northwind)* adatbázis szinten. Ezt csak azért tette, hogy a tranzakció alatt ne forgassák fel alapjaiban az adatbázist, ám semmi más zárolás nem látszik.
Sajnos azt nem látjuk, hogy a kiolvasott sorokat még a SELECT lefutása idejére sem zárolta a szerver, mert mire a végrehajtás az sp_lock-ra kerül, a zárolások *(ha lettek volna)* már rég megszûntek volna. Azt azonban könnyû megfigyelni a következô példában, hogy ezen a szinten lehet nem véglegesített *(csúnya hunglish-el élve nem kommitált)* lapokat olvasni, és hogy a SELECT nem vár az exkluzív zárolások miatt. Futtassuk le az alábbi kódot egy másik Query Analyzer ablakban:

```
BEGIN TRAN

UPDATE
  Employees
SET
  LastName = 'Borzaska'
```

Azaz megkezdünk egy tranzakciót, amiben minden alkalmazott családi nevét Borzaskára állítjuk. A tranzakciót logikailag még nem véglegesítettük, ám a változások fizikailag már rögzítôdtek a táblába. Mit lát ebbôl a korábbi lekérdezésünk *(READ UNCOMMITTED szinten)*?

```
LastName             FirstName
-------------------- ----------
Borzaska             Nancy
Borzaska             Andrew
...
```

Azaz látja a beírt, de még nem véglegesített adatokat! Ezért hívják dirty read-nek ezt a szintet. Viszont láttuk, hogy nem tudtuk megakadályozni az olvasást még egy egész táblára szóló UPDATE-el sem, azaz ezen a szinten az adatbázist olvasó mûveletek nem foglalkoznak még az Exclusive lock-okkal sem.
Hogy megnyugodjanak a kedélyek, görgessük vissza az elôbbi félbehagyott tranzakciónkat:

```
ROLLBACK TRAN
```

2. READ COMMITTED

Ez az alapértelmezett elszigeteltségi szint az SQL szerverben. Ezen a szinten a SELECT utasítások Shared lock-okat helyeznek el azokon a sorokon, amelyeket éppen olvasnak. Emlékezzünk vissza, a Shared lock egy olyan zárolási típus, amit akárhányan olvashatnak, de senki nem írhat. Azaz a Shared lock megakadályozza, hogy valaki belenyúljon azokba az adatokba, amit a SELECT éppen olvas. Amint a megfelelô sor, lap vagy tábla kiolvasása megtörtént, a zárolások feloldódnak. Amennyiben a SELECT halad elôre a sorok olvasásával, és beleütközik egy Exclusive lock-ba, ami azt mondja neki, hogy állj, ne tovább, akkor kénytelen arra várni, hogy az Exclusive lock feloldódjon. Ellenkezô esetben visszalépnénk az elôzô szintre, és olyan adatokat olvasnánk, amelyeket még nem véglegesítettek. Ez a szint azonban arról szól, hogy csak olyan adatokat olvashat az adatbázisból, amelyeket már véglegesítettek, innen a szint neve is. Azaz ezen a szinten logikailag mindig konzisztens adatokat olvasunk ki.
Futtassuk le a korábbi teszt tranzakciónkat ezen a szinten is:

```
BEGIN TRAN
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
...
```

A tranzakciót egyedül lefuttatva a lekérdezés kimenete és a keletkezett zárolások pont úgy néznek ki, mint az elôzô szinten. Azonban gyökeresen más a helyzet, ha elindítjuk a másik „zavaró” tranzakciónkat is. Azaz futtassuk le az abban található UPDATE-et, de ne hajtsuk végre a ROLLBACK TRAN-t, hanem helyette indítsuk el az elsô lekérdezést!
Mit látunk? Semmit. A lekérdezés csak fut, csak fut... Mivel a másik tranzakció adatmódosítása Exclusive lock-okat helyezett el a tábla sorain *(sôt ez egész táblán, mert minden sort módosítottunk)*, a SELECT ezen a szinten már figyelembe veszik ezeket a zárolásokat, és addig nem hajlandó kiolvasni az adatokat, amíg a zárolás el nem takarodik a sorokról. Ehhez hajtsuk végre a ROLLBACK TRAN-t a második tranzakcióban! Ekkor az elsô tranzakció is befejezi a futását, és kiadja az eredeti, módosítás elôtti sorokat:

```
LastName             FirstName
-------------------- ----------
Buchanan             Steven
```

Amennyiben a második tranzakció nem visszagörgeti, hanem érvényesíti a tranzakciót COMMIT TRAN-al, természetesen akkor is folytatja a futást az elsô tranzakció SELECT-je, csak a már módosított adatokat olvasva.
Ezen a szinten semmi nem biztosítja azt, hogy a tranzakción belül ugyanazokkal a feltételekkel visszaolvasva az adatokat ugyanazt az eredményt kapjuk két különbözô idôpillanatban. Lehet, hogy más tranzakció megváltoztatja az általunk kiolvasandó sorokat a két kiolvasás között, ezt nevezzük nem megismételhetô olvasásnak *(non-repeatable read)*. Az is elôfordulhat, hogy beszúrnak olyan sorokat a két SELECT közötti idôben, amelyek megjelennek a második SELECT eredményhalmazában. Ezeket a megjelent sorokat hívják fantomoknak *(phantoms)*. A következô két szint ezeket a problémákat fogja orvosolni.

3. REPEATABLE READ

Itt már biztosak lehetünk abban, hogy logikailag helyes adatokat olvashatunk ki a táblákból, plusz, hogy egy tranzakción belül ugyanazt az olvasást többször megismé-

telve mindig ugyanazt az eredményt kapjuk vissza. Ezt azt jelenti, hogy a már olvasott sorok tartalma nem fog megváltozni, de nem jelenti azt, hogy nem jelenhetnek meg új sorok más tranzakciók ármány munkájának köszönhetôen. Hogyan védekezik az SQL Server a már olvasott sorok módosítása ellen? Úgy, hogy a SELECT-ek által végigolvasott sorokra *(lapokra vagy táblára)* elhelyezett Shared lock-okat nem oldja fel egészen a tranzakció végéig. Ezek után hiába akarja valamelyik másik tranzakció módosítani a már leválogatott sorokat, a Shared lock-ok nem engedik meg, hogy megtegye, egészen a tranzakció befejezéséig.
Ezen a szinten már nagyon erôsen érezhetô a zárolások miatti párhuzamosság csökkenése, hisz egy

```
SELECT * FROM tábla
```

utasítással gyakorlatilag befagyasztjuk az összes olyan tranzakciót, ami a táblán akar módosítani. Azaz csak tényleg olyankor érdemes bevetni, amikor a tranzakción belül többször ki kell olvasni ugyanazokat a sorokat, és fennáll a veszélye, hogy valaki közben módosítja ôket.
Ha megnézzük, milyen zárolások keletkeznek ezen a szinten, akkor a következôt látjuk:

```
spid dbid ObjId       IndId Type Resource       Mode
---- ---- ----------- ----- ---- -------------- ----
51   6    0             0    DB                  S
51   6    1977058079    1    KEY  (0500d1d065e9) S
51   6    1977058079    2    KEY  (6c01b4c53be8) S
51   6    1977058079    1    PAG  1:136          IS
51   6    1977058079    0    TAB                 IS
51   6    1977058079    2    PAG  1:385          IS
```

Azaz a lock manager elhelyez Shared lock-okat sorokra a kulcsaikon keresztül *(2. és 3. sor)*, valamint Intent Share lock-okat a lekérdezett sort tartalmazó lapokra *(4. és 6. sor)*, valamint a táblára *(5. sor)*. Az Intent Share jelzi más tranzakcióknak, hogy ne is próbáljanak Exclusive lock-ot kérni a kérdéses lapokra vagy az egész táblára, mert úgysem fog sikerülni, hisz az adott „nagy" tartományokon belül vannak olyan sorok, amelyek Shared lock-kal védettek.
Miért van két sor és lap zárolás, amikor a lekérdezés kimenete csak egy sort tartalmaz? Láthatjuk, hogy különbözô indexekhez *(IndId oszlop)* tartoznak a zárolások. Az Employees táblán három index is van, ezek közül kettôt használt a lekérdezés. A LastName-re szûrtünk, ehhez a LastName oszlopra definiált Nonclustered index-et használta a szerver *(ez könnyen ellenôrizhetô a végrehajtási terv megtekintésével is)*. Miután megtalálta a LastName index táblában a megfelelô sorokat *(jelen esetben 1 sor)*, a Nonclustered index, mint sorazonosító segítségével kiolvassa a megfelelô sor tartalmát. Ahhoz, hogy biztosítsa a zárolást bármelyik indexet használó tranzakció elôl, kénytelen zárolni mindkét index által lefoglalt sorokat és lapokat.

4. SERIALIZABLE

Nagyon hasonlít a REPEATABLE READ szintre, csak itt meg kell akadályozni azt is, hogy ugyanazt a SELECT-et megismételve új sorok jelenjenek meg az eredményhalmazban. Ehhez a szervernek le kell zárolni a teljes lehetséges tartományt, amelyet a SELECT WHERE feltétele jelöl ki. Az SQL Server a tartomány zárolására a már említett key-range lock-ot használja.

Nézzük meg az elôbbi lekérdezést, aminek feltétel része a következô volt:

```
WHERE
  LastName LIKE 'B%'
```

E szint logikájának megfelelôen a szervernek le kell zárolnia az összes olyan lehetséges index irányokat, amelyeken keresztül B betûvel kezdôdô nevû sorokat be lehetne szúrni a táblába. Milyen zárolások generálódnak ennek érdekében? *(az objid oszlopot nyomdai okokból kihagytam)*

```
spid dbid IndId Type Resource       Mode
---- ---- ----- ---- -------------- ----
54   6    0     DB                  S
54   6    0     TAB                 IS
54   6    1     PAG  1:99          IS
54   6    2     PAG  1:97          IS
54   6    2     KEY  (7901573565c0) RangeS-S
54   6    255   PAG  1:225         IS
54   6    255   RID  1:225:12      S
54   6    1     KEY  (0500d1d065e9) S
54   6    255   RID  1:225:11      S
54   6    2     KEY  (6c01b4c53be8) RangeS-S
```

Látható, hogy a két RangeS-S *(Shared Key-Range and Shared Resource)* zárolás lezárta a LastName-re definiált Nonclustered index két végét *(A és C betûvel kezdôdô sorok közötti rész)*, így oda nem lehet új sorokat beszúrni.
A szintek tárgyalásánál nem szóltam az sp_lock kimenetébôl az utolsó oszlopról. Abban látható, hogy a zárolást megkapta-e a kérô, vagy csak vár rá. Az összes példámban a mezô értéke GRANT volt, azaz a kérô megkapta a zárolását. A READ COMITTED szintnél az UPDATE tranzakció blokkolja az olvasni kívánó tranzakciót, ilyenkor az utolsó oszlopban WAIT olvasható, azaz vár arra, hogy a másik tranzakció feloldja az általa foglalt zárolást.

Locking hints

Többször hivatkoztam arra, hogy az SQL Servert lehet befolyásolni abban, hogy milyen típusú, és milyen finomságú zárolásokat helyez el a tranzakciók során érintett adatokon. Most jött el az ideje, hogy áttekintsük ezeket.
A SELECT, UPDATE, DELETE és INSERT utasításokat ki lehet egészíteni egy WITH *(hint)* záradékkal, amely segítségével az SQL Servert el lehet téríteni az általa választott mûködéstôl, és így megszabhatjuk, hogy milyen index-et, zárolást satöbbi használjon a táblák elérése során. Mi itt, most csak a zárolásokat befolyásoló hint-ekkel foglalkozunk.
Az elsô csoport a zárolás finomságára vonatkozik. A ROWLOCK arra utasítja a szervert, hogy a zárolandó sorok számától függetlenül *(még ha az egész táblára is vonatkozik)* ne használjon nagyobb kiterjedésû zárolást, mint sor szintût. Hasonlóan a PAGLOCK, TABLOCK lap illetve tábla szintû zárolás használatára kéri a szervert.

Példa:

```
SELECT * FROM Orders WITH (PAGLOCK)
WHERE OrderID = 1213
```

Az elôbbi módosítók a zárolás finomságát állították. A következôk a zárolás típusát szabályozzák.
Az UPDLOCK segítségével a SELECT az alapértelmezetten használt Shared lock helyett Update lock-ot helyez el az olvasott táblán. Ennek elônye, hogy a már olvasott sorokon a tranzakció végéig megmarad az Update lock, így mások olvashatják az általunk kiolvasott sorokat, de nem módosíthatják azokat. *(Az Update és a Shared lock között annyi a különbség, hogy a már fennálló Shared lock-ra kiadható egy Update lock, de egy Update-re egy másik Update már nem.)*
Az XLOCK Exclusive lock-ot helyez el az adott utasítás által érintett sorokon. Azaz például egy ilyen módon átidomított SELECT képes exkluzívan zárolni egy egész lapot vagy táblát.
A NOLOCK és a READUNCOMMITTED ugyanazt jelenti, azaz mindenféle zárolástól függetlenül felolvassa a kért adatokat. Ezt kiadva a tranzakció összes utasítására ugyanazt érjük el, mint ha a tranzakció elszigeteltségi szintjét az elején READ UNCOMMITTED-re állítottuk volna. Gyakori felhasználás statisztikákban:

```
SELECT OrderID, SUM(Amount*UnitPrice)
FROM [Order Details] WITH (NOLOCK)
GROUP BY OrderID
```

Azaz az Order Details táblán mûködô egyéb adatmódosító tranzakcióktól függetlenül, mindenféle zárolást kikerülve olvasunk adatokat.
A HOLDLOCK és a SERIALIZABLE lock hint-ek hatására az SQL Server úgy kezeli az érintett táblákon a zárolásokat, mintha a tranzakció SERIALIZABLE módban lenne, azaz a Shared lock-okat nemcsak az olvasás idejére, hanem az egész tranzakció idejére fenntartja a már olvasott sorokon *(innen a HOLDlock név)*.
A READCOMMITTED hint a READ COMMITTED elszigeteltségi szint párja. Mivel ez az alapértelmezett szint, ritkán van szükség rá, hogy explicit kiírjuk.
Hasonlóan a REPEATABLEREAD az azonos nevû izolációs szint párja.
Az utolsó hint egy kicsit más, mint az elôzôek. A READPAST azt mondja egy SELECT utasításnak, hogy egyszerûen ugorja át azokat a sorokat, amelyeket más tranzakció zárolt, és olvassa fel a nem zárolt sorokat. Ez csak READ COMMITTED elszigeteltségi szintû tranzakciókban mûködik, és csak a sor szintû zárolásokat tudja átlépni. Egy adatbázis elmélettel foglalkozó embernek ettôl égnek áll a haja, de a való életben vannak olyan helyzetek, amikor hasznos lehet ez a szolgáltatás.
Azt írtam, hogy ezeket a hint-eket mind a négy alaputasítással lehet használni. Természetesen ez csak korlátozottan igaz, hiszen például a READPAST-nak nincs értelme az adatmódosító utasításoknál, azaz csak SELECT-el használható. Mindegyik hint-nek megvan a maga logikája, és csak azokon a helyeken mûködik, ahol van értelme.

Application lock-ok

SQL Server 2000 újdonság az application lock-ok megjelenése. Segítségükkel létrehozhatunk saját zárolási mechanizmusokat a szerver lock manager-ének felhasználásával. Láttuk a zárolások finomságának tárgyalásánál, hogy a lock manager az igazából nem tud róla, hogy ô milyen objektumon végez zárolást *(a nevét tudja, de a belsô struktúrájáról semmit nem tud)*, csak van neki egy táblázata, amely alapján eldönti, hogy az ütközô zárolás kérések esetén továbbengedheti-e az igénylôt, vagy várakoztatnia kell, amíg elfogynak a konkurens zárolások. Most megkaptuk ezt a logikát, amely segítségével más programnyelveken megszokott kritikus szekciókat illetve szemaforokat valósíthatunk meg az alkalmazásainkban.
Saját zárolás létrehozása nagyon egyszerû. Az sp_getapplock tárolt eljárás meghívásával kérünk egy általunk megálmodott zárolási típust, egyedi néven. Elindítjuk a védendô, zárolandó eljárásunkat. Az eljárásunk lefutása után az sp_releaseapplock eljárással szabadíthatjuk fel a zárolást.
Gyakori feladat például az, hogy egy tárolt eljárást egyszerre csak egy felhasználó futtathat. Application lock-ok felhasználásával ezt nagyon egyszerûen megoldhatjuk:

```
EXEC sp_getapplock 'SociLock', 'Exclusive',
'Session'

EXEC VédendôTároltEljárás

EXEC sp_releaseapplock 'SociLock', 'Session'
```

Az sp_getapplock elsô paramétere a zárolás egyedi neve. A második paraméter a zárolás típusa, amit mi most Exclusive-ra állítottunk, mert azt akarjuk, hogy miután valakinek sikerült túljutni a záráson csak ô futtathassa a VédendôTároltEljárás-t, egészen addig, míg az sp_releaseapplock-al el nem engedjük a zárolást. A 'Session' azt jelenti, hogy ugyanarról a felhasználói kapcsolatról nem hatásos a zárolás, csak különbözô kapcsolatok között. Ez azt is jelenti, hogy ugyanaz a felhasználó többször is lefuttathatja a védett eljárást, mert a lock saját magára hatástalan. Ha azt akarjuk, hogy még ugyanaz a felhasználó se futtathassa többször a közbensô eljárást, akkor a 'Session' helyett 'Transaction'-t kell írni, és a három eljáráshívást tranzakcióba (BEGIN TRAN, COMMIT TRAN) kell foglalni. Ekkor a zárolás tranzakciószintû lesz, így még egyazon felhasználói kapcsolaton futó párhuzamos tranzakciók is zárolják egymást, megakadályozva a párhuzamos futtatást.
A Shared és az Exclusive és a többi zárolási típus variálásával kialakíthatunk más jellegû zárolási sémákat is, amelyek megfelelôen támogatják az alkalmazásunk logikáját.

Zárszó

Cikksorozatunk eddigi legnehezebb része volt a zárolások témaköre. Ezután már általában könnyebb, gyakorlatiasabb részek jönnek. Úgyhogy az a kedves olvasó, akinek volt türelme végigolvasni és értelmezni a cikket (abban már csak reménykedni merek, hogy az esetleges kérdôjeles részekhez elôkerült a Books Online is), már megtette az elsô lépést abban az irányba, ami a professzionális adatbázis tervezés felé vezet. Az adatbázis zárolási eljárásának ismerete nélkül tranzakciókat és adatbázisokat tervezni vakrepülés, amely elôbb-utóbb egy sziklafalon végzôdik.
A következô cikkbe szorult át a dead-lock-ok elmélete és gyakorlata, amely azonban csak a zárolások ismeretében érthetô meg. Visszavárom Önöket a halálos ölelések szigetén, a következô számban!

Soczó Zsolt MCSE, MCSD, MCDBA
Zsolt.Soczo@netacademia.net

Transact SQL *(VII. rész)*

Az árvíztûrô tükörfúrógép

Bevezetés

Egy hónap kihagyás után újra itt a Transact SQL sorozat, töretlen lendülettel! E havi részünkben azokkal a nyelvi finomságokkal és SQL Server 2000 újdonságokkal foglalkozom, amelyek nem kaptak akkora hírverést mint mondjuk az XML támogatás, de nagyon fontos apróságok, amelyek különösen magyar nyelvû szövegeket kezelô programok írásakor nagyon fontosak. Járjunk utána az ô û betûk misztériumának!

Az alapprobléma

Ki ne bosszankodott volna azon, hogy egy lementett *(backup)*, magyar nyelvi beállításokkal mûködô adatbázist nem lehetett visszaállítani egy angol nyelvi beállításokkal telepített másik SQL Server 7-re, a nyelvi beállítások különbözôsége miatt? Ennek az az egyszerû oka volt, hogy az SQL Server 7-ben a nyelvi beállítás globális, kiszolgálószintû volt, amely az összes adatbázisra, és azon belül az összes objektumra, adatra, oszlopra vonatkozott. Ez az információ benne volt a felhasználói adatbázisokban is, így visszaállításkor az SQL Server 7 észrevette a nyelvütközést, és nem engedte a visszaállítást.

De miért van egyáltalán szükség az egész nyelvi hókuszpókuszra? Miért volt ez annyira beleépülve a kiszolgálóba? A probléma alapja a nemzetek nyelveinek különbözôségében keresendô.

Ha egy karaktert egy bájton ábrázolunk, akkor 256 féle karaktert tudunk leírni. Ez bôven elég az angol abc leírásához, még sok egyéb extra karakter *(+!%...)* is belefér. Azonban a bôségbôl gyorsan „szûkség” lesz, ha elkezdjük felmérni és megpróbáljuk ábrázolni az összes nemzet valamennyi karakterét. 256 hely erre nem elég. Ezt feloldandó minden ország, helyesebben minden országcsoport, amelynek azonos karaktereik vannak az abc-ben kap egy karakter kódtáblát, kódlapot *(character set)*, ami az ô nyelveikben található betûket rendeli hozzá a 0...255-ös tartományhoz. A nem ékezetes betûk általában ugyanarra a kódra vannak leképezve a legtöbb kódlapban, így például a kis „a” betû a 97. pozícióra. Ellenôrzés:

```
-- Nyelvi beállítás: Latin1_General
-- (nyugati nyelvek)
SELECT ASCII('a')
97
```

Az ékezetes betûk helye már legtöbbször nyelvfüggô. Bizonyára mindenkinek ismerôs a magyar „ô” és az „û” betûk problémája. Ha egy számítógépen a nyelvi beállítások nem megfelelôek, akkor mindig ezzel a két karakterrel szokott baj lenni. Nézzünk csak ennek a körmére! Latin 1 *(nyugati nyelvek)* kódkészletet használva kérdezzük le az „ô” betû kódját, azaz azt, hogy ez a betû a Latin 1-es kódlap melyik pozícióján van értelmezve?

```
--Adatbázis beállítás: Latin1_General
SELECT ASCII('ő')
111
```

Ez egy picit gyanús, mert az ékezetes betûk általában a 128 feletti pozíción foglalnak helyet. Tegyünk egy ellenpróbát, az így kapott karakterkódot alakítassuk vissza az adott kódlapon érvényes karakterré:

```
SELECT CHAR(ASCII('ő'))
o !!!
```

Hoppá, a kis „ô” betûnkbôl „o“ lett! Nem véletlenül volt gyanús a 111-es kód. Az valójában az „o” betû kódja:

```
SELECT ASCII('o')
111
```

Mi volt itt a gond? Az, hogy a Latin1-es kódlapban, nincs helye a mi „ô” betûnknek! Nézzük csak meg milyen karaktereket ismer a Latin1:

```
DECLARE @i int
DECLARE @s varchar(2000)
SET @s = ''
SET @i = 32
WHILE @i < 256
BEGIN
  SET @s = @s + CHAR(@i)
  SET @i = @i + 1
  --Sortörés minden 32. karakter után
  IF @i % 32 = 0 SET @s = @s + CHAR(13)
END
PRINT @s
```

A lekérdezés kimenetét a nyomdai utómunkák okozta lehetséges karakter konverziók, torzulások miatt grafikusan mutatom meg. Hiába, a karakterkonverzió nem csak az SQL Serverben problémás pont. :-)

Karakterek a Latin1 kódlapban. Hová lettek az ô û betûink?

Karakterek a magyar kódlapban. Itt már a helyükön vannak az ékezetes betûink.

Jól látható, hogy kétvesszôs „ô” nincs ebben a készletben, csak kalapos, meg hullámos tetejû. Azaz, ha szeretnénk használni normális magyar karaktereket, akkor át kell kapcsolnunk az adatbázisunkat egy olyan kódkészletre, amely ismeri a rendes betûinket. Praktikusan magyarra. Lássuk, ekkor jól mûködnek-e a konverziók:

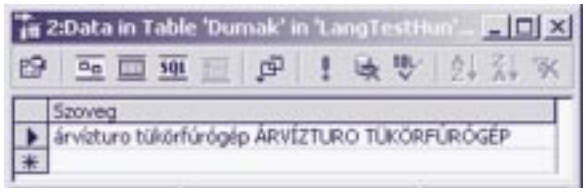
```
--Átkapcsolunk magyarra
ALTER DATABASE LangTest
COLLATE Hungarian_CI_AI

--Teszt 1: karakterből ASCII kód
SELECT ASCII('ő')
245

--Teszt 2: a kapott ASCII kódból karakter
SELECT CHAR(ASCII('ő'))
ő
```

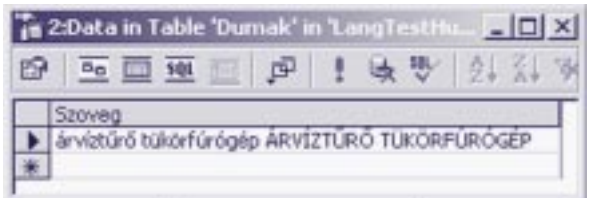
Remekül megy! Azaz leszögezhetjük, hogy a magyar karakterek sikeres kezeléséhez vezetô út elsô lépése a megfelelô karakterkészlet beállítása az adatbázisra.
Az összehasonlítás kedvéért nézzük még meg a magyar kódkészletet is:

Amikor két számítógép, vagy két adatbázis között mozgatunk nem UNICODE szöveges adatokat, akkor a célhelyen kódkonverzió történik. Ez a konverzió a célgép operációs rendszerének kódtábláiban van definiálva. Abban az esetben, ha a forrásszövegben van olyan karakter, ami a célszerver *(adatbázis)* kódtáblájában nincs definiálva, a konverzió sokszor az adatok megváltozásával jár. Ennek tipikus megjelenése, amikor egy adatbázis ügyfélalkalmazás nem jeleníti meg az „ôû” betûket, hanem „ou“-t ír ki helyette. Például a képen látható Enterprise Manager ablakban megjelenítettem egy tábla tartalmát, amelyben helyesen, magyar kódlappal vannak eltárolva a karakterek. Csakhogy az Enterprise Managert futtató Windows 2000 System Local-je English-re volt állítva, így az ügyféladatbázis meghajtóprogramja átfordítja a kiszolgálóról letöltôdô karaktereket angol kódlapra – megkérdôjelezhetô sikerrel:



Hová lettek az ôû betûûûk?

Ha a System Local-t magyarra állítom, akkor mindjárt jól mûködnek az ékezetes karakterek is:



Ügyfél Windows 2000 System Locale: magyar. Megjöttek az ôû betûk.

Nem állítom, hogy át kell állítani mind az SQL Server mind az ügyfél munkaállomások System Local-jét magyarra ahhoz, hogy ne történjék adatvesztés a konverzió során, de ha más ellenérv nem szól ellene, akkor érdemes ily módon beállítani ôket. Ez nem feltétlenül szükséges, viszont elégséges feltétele a fájdalommentes, magyar szövegekkel operáló adatbázismûveleteknek.

Adatbázis collation

Lehet, hogy már sikerült meggyôzni a cégvezetést, és minden gépen magyar beállításokkal mûködik az NT vagy a Windows 2000. Biztos lehetek benne, hogy ezek után minden rendben lesz az ékezetekkel? Természetesen – nem! Az SQL Server 2000-ben adatbázisonként is megadható a nyelvi beállítás, így elôfordulhat, hogy mégis gond lesz az adatmozgatások során. Nézzünk egy példát, ami jobban megvilágítja a jelenséget! Hozzunk létre két adatbázist, az egyiket állítsuk magyar collation-ûre, a másikat Latin1-re. Szúrjuk be a cikk címében szereplô tesztszöveget a magyarra állított adatbázis egy táblájába, majd egy közönséges INSERT … SELECT párossal másoljuk át a magyarba beszúrt sort az angol adatbázis táblájába. Figyeljük meg, mikor történik konverzió, és mikor nem! A példában a System Local magyar mind a kiszolgálón, mind a munkaállomáson *(egy gépen vannak :)*, tehát az nem okozhat problémát.

```
USE master
--Angol (Latin1) COLLATION-ű adatbázis létrehozása
CREATE DATABASE LangTestEng
ON
(
  NAME='LangTestEngPrimary',
  FILENAME='c:\temp\langtesteng.mdf'
) COLLATE Latin1_General_CI_AS

--Magyar COLLATION-ű adatbázis létrehozása
CREATE DATABASE LangTestHun
ON
(
  NAME='LangTestHunPrimary',
  FILENAME='c:\temp\langtesthun.mdf'
) COLLATE Hungarian_CI_AS

--Teszttáblák létrehozása mindkét adatbázisban
USE LangTestHun

CREATE TABLE Dumak (
  Szoveg varchar(50) )

USE LangTestEng

CREATE TABLE Dumak (
  Szoveg varchar(50) )

USE LangTestHun

--Beszúrás a "magyar" adatbázis táblájába
INSERT Dumak (Szoveg) VALUES (
'1. árvíztűrő tükörfúrógép ÁRVÍZTŰRŐ TÜKÖRFÚRÓGÉP'
)

--Jól sikerült?
SELECT Szoveg FROM Dumak
```

Kimenet:

```
Szoveg
--------------------------------------------------
1. árvíztűrő tükörfúrógép ÁRVÍZTŰRŐ TÜKÖRFÚRÓGÉP
```

Azaz a magyar beállításokkal rendelkezô táblába sikerült helyesen beszúrni a szöveget. Mehet a másolás.

```
--Másoljuk át a magyarba beszúrt sort
--az angol collation-ű adatbázisba.
INSERT LangTestEng..Dumak SELECT Szoveg FROM Dumak

--Ellenőrizzük le, mi jelent meg benne!
SELECT Szoveg FROM LangTestEng..Dumak
```

Az másolatban bizony leestek a kedvenc kétvesszôs magyar karaktereink:

```
Szoveg
--------------------------------------------------
1. árvízturo tükörfúrógép ÁRVÍZTURO TÜKÖRFÚRÓGÉP
```

Hasonlóan kiábrándító az eredmény, ha közvetlenül az angol nyelvû adatbázis táblájába próbálunk adatokat csempészni:

```
USE LangTestEng

-- Beszúrás közvetlenül az angol
-- adatbázis táblájába
INSERT Dumak (Szoveg) VALUES ('2. árvíztűrő tükörfúrógép ÁRVÍZTŰRŐ TÜKÖRFÚRÓGÉP')

--Jól sikerült?
SELECT Szoveg FROM Dumak
```

Kimenet:

```
Szoveg
--------------------------------------------------
1. árvízturo tükörfúrógép ÁRVÍZTURO TÜKÖRFÚRÓGÉP
2. árvízturo tükörfúrógép ÁRVÍZTURO TÜKÖRFÚRÓGÉP
```

Oszlop collation

Ha létrehozunk egy táblát egy adatbázisban, akkor annak a szöveges oszlopai öröklik a szülô adatbázis collation-jét. Ha ez nekünk nem megfelelô, akkor a tábla létrehozásakor akár oszloponként megadhatunk különbözô collation-öket. Így elôállhat az a helyzet, hogy egy angol collation-û adatbázisban létrehozunk egy magyar nyelvû oszloppal felvértezett táblát. Például:

```
USE LangTestEng

CREATE TABLE Dumak (
  Szoveg varchar(50) COLLATE Hungarian_CI_AS )
```

Ha a korábbi példában látott módon a magyarból másolunk az angolba, akkor most helyesen fognak átmenni az ékezetek, mert a céloszlop collation-je megegyezik a forráséval, így nincs szükség konverzióra.
Ami viszont nagyon érdekes, hogy a közvetlenül az angol nyelvû adatbázisba beszúrt adatokról elvesznek az ékezetek, annak ellenére, hogy a céloszlop magyar nyelvû:

```
USE LangTestEng
INSERT Dumak (Szoveg) VALUES ('3. árvíztűrő tükör-
fúrógép ÁRVÍZTŰRŐ TÜKÖRFÚRÓGÉP')

SELECT Szoveg FROM Dumak
Szoveg
-----------------------------------------------
3. árvízturo tükörfúrógép ÁRVÍZTURO TÜKÖRFÚRÓGÉP
```

Ez még ügyféloldalon konvertálódik át „éktelenné”, ami megelôzhetô, ha a tesztszövegünket megjelöljük *(N a szöveg elôtt)*, hogy az UNICODE formátumú, így az konverzió nélkül át tud utazni a kiszolgálóra. Igaz, hogy ott vissza kell konvertálni egybájtossá, de azt már az SQL Server helyesen, az oszlop típusának megfelelôen teszi meg. Azaz:

```
USE LangTestEng

--Beszúrás közvetlenül az angol
-- adatbázis táblájába
INSERT Dumak (Szoveg) VALUES (N'3. árvíztűrő tükör-
fúrógép ÁRVÍZTŰRŐ TÜKÖRFÚRÓGÉP')

SELECT Szoveg FROM Dumak

Szoveg
-----------------------------------------------
3. árvíztűrő tükörfúrógép ÁRVÍZTŰRŐ TÜKÖRFÚRÓGÉP
```

Láthatjuk, hogy sok apró finomság állhat az ékezetek útjába. Mit tehetünk a siker érdekében? Ha van lehetôségünk homogén SQL Server-farm kialakítására, és nem szeretnénk csak magyar nyelvet, esetleg angolt *(részhalmaza a magyar kódlapnak, általában nincs vele probléma)* használni a táblákban, akkor állítsuk be mind az NT, 2000 kiszolgálókat, mind az SQL Servereket magyarra, és hagyjuk, hogy az adatbázisok és az oszlopok örököljék a magyar beállításokat. Ez különösen hálás lesz akkor, ha adatokat kell replikálni a szerverek között. A replikáció nem szereti a vegyesen beállított collation-öket, úgyhogy megéri még az elején konszolidálni a kiszolgálóparkot.

UNICODE, az univerzális megoldás?

Most mondhatná valaki, hogy miért kell ennyit problémázni a nemzeti karaktereken, amikor már ezer éve kitalálták a UNICODE-ot? Valóban, a UNICODE arról szól, hogy ne egy, hanem két byte-on írjuk le a karaktereket, így az eredô 65536-os tartományba majd csak belefér minden ország összes karaktere. Ez így igaz. Azonban nagy tömegû adathalmaznál ennek ára van – a kétszeres helyfoglalás. 100 MByte-nál ez nem kérdés, de pár tíz GByte felett ez már nagyon is számít.
Emellett a UNICODE túl nagyágyú, ha tudjuk, hogy soha nem fogunk ugyanabban az oszlopban többféle nyelvû szövegeket tárolni. Amennyiben ez a feladat, gondolkodás nélkül az UNICODE-hoz kell nyúlni. Egyes ázsiai nyelveknek több ezer karakteres betûkészlete van, ezeket csak UNICODE karakterekkel lehet leírni - ebbe is ritkán fut bele magyar ember. Ha SQL Serverek között mozgatunk adatokat, és nem azonos kódlapokat használnak a kommunikáló felek, akkor is érdemes UNICODE-ot használni, így várhatóan nem lesz gond a karakterek átvitelével, mert nincs szükség kódfordításra.
Azaz, ha teljesen biztosra akarunk menni, és kiszolgáló-, illetve adatbázisbeállítástól független módon akarunk magyar esetleg más nyelvû szövegeket tárolni, akkor használjunk UNICODE karaktereket:

```
USE LangTestEng

CREATE TABLE DumakUNI (
  Szoveg nvarchar(50) )
```

Ekkor nincs szükség karakterkonverzióra, hisz UNICODE esetén nincs szükség kódlapokra, a 65536-os tartományban jól megférnek egymás mellett a nemzetek karakterei. Egyre figyeljünk nagyon, használjuk az N prefixet a konstans szövegek elôtt, jelezve, hogy UNICODE adatról van szó. Így egy angol nyelvû oszlopba, egy német alapértelmezett nyelvû kiszolgálón is hibátlanul át fognak menni a magyar ékezetes karakterek *(is)*.
Mindezen elônyök ellenére, ha nincs kényszerítô okunk a UNICODE használatára, érdemes maradni az egybájtos típusoknál, megfelelô kódlap kiválasztásával. Ez persze vitatható pont, ez csak az én személyes álláspontom.

Collation – részleteiben

Mit takar az az immáron többször is felbukkanó homályos fogalom, hogy nyelvi beállítás, vagy collation? Alapvetôen a szöveges információk tárolását, leképezését *(kódlap)*, sorbarendezését és a karakterek viselkedését az összehasonlítások során határozza meg. Járjuk ezt egy kicsit körbe!
Kezdjük az összehasonlításokkal. „Alma“ = „alma“? Ha Case Insensitive módon hasonlítjuk ôket össze, akkor egyformának tekinthetjük ôket, azaz a kis-nagybetû különbségeket nem vesszük figyelembe. Ha a beállítás Case Sensitive, akkor természetesen a kettô nem egyezik meg. Ez eddig nem volt nehéz.
Amit már sokkal kevesebben ismernek, az az Accent Sensitivity fogalma. Ez azt jelenti, hogy két szöveg, amelyek csak ékezetben különböznek egymástól egyformának tekinthetô-e? Például „Eger“ = „Egér“? Ha a nyelvi beállítás Accent Insensitive, akkor igen. Ha Accent Sensitive, akkor különböznek egymástól. Mi magyarok, akik ékezetes betûkkel írunk általában megkülönböztetjük az ékezetes betûket éktelen társaiktól.
Japán nyelvet kedvelôknek: ha az adott collation Kana-sensitive, akkor a Hiragana és Katakana karaktereket megkülönbözteti a kiszolgáló. Hurrá! Ez nagyon hasznos szolgáltatás a Japánoknak – na de miért kell ezzel nekünk törôdnünk?

Azok, akik próbáltak már visszaállítani SQL Server 7-es adatbázist, tudják, hogy érdemes tudni a szerver Kana érzékenységének a beállított értékét, még akkor is, ha az a magyar nyelvre nem releváns. Ugyanis egy 7-es adatbázisban benne van az összes nyelvi beállítás aktuális értéke, beleértve a Kana-t is, valamint még egyet, amirôl nem beszéltem, a Width érzékenységet is. Mit tennénk abban az esetben, ha kapunk egy SQL Server 7-es backup-ot, és azt mondják, hogy telepítsünk egy új SQL Servert, és állítsuk vissza rá a mentést? Ha tudjuk a Server eredeti nyelvi beállításait, beleérve kanna és vid pajtásokat is, akkor egyszerûen feltelepítjük a szervert, és visszaállítjuk a kért adatbázist. Viszont, ha nem, akkor maximum 16 féleképpen *(Case, Accent, Kana, Width kombinációi)* kell feltelepíteni, és amelyikre visszatöltôdik az adatbázis, úgy volt beállítva az eredeti korábban. A rebuildm.exe meggyorsíthatja ezt a folyamatot, amivel az SQL Server teljes újratelepítése nélkül át tudjuk változtatni az alapértelmezett nyelvi beállításokat, de még így is elég gyilkos móka a restore.

Tipp: ha van a közelben egy SQL Server 2000, akkor arra probléma nélkül vissza lehet tölteni az SQL 7-es adatbázist, és meg lehet nézni az nyelvi beállítását.

Azaz látjuk, hogy SQL Server 7 esetén telepítéskor meg kellett adni a karakterkészletet, ami „beleégett" a szerverbe, és attól kezdve csakis azzal a kódkészlettel tudott dolgozni. Ez azt jelentette, hogy onnantól kezdve az összes, nem UNICODE oszlop a megadott kódlappal, az operátorok és sorbarendezések pedig a beállított nyelv logikája szerint mûködtek. Ha más nyelv kellett, újra kellett építeni a master adatbázist, ami hatásaiban egy SQL Server újratelepítéssel egyenértékû.
SQL Server 2000 esetén három szinten jelennek meg a nyelvi beállítások: szerverre alapértelmezetten, adatbázisszinten és a táblákban oszlopszinten. A Server alapértelmezett beállítása lesz érvényes az összes rendszeradatbázisra is: master, model, tempdb, msdb, és distribution. Ezt csak a telepítés folyamán lehet megadni, késôbb már csak újratelepítéssel cserélhetô le másra. Ez kihat az összes objektum nevének és egyéb tulajdonságának a kezelésére is. Így az oszlopnevekre is, amibôl mókás hibák adódhatnak. Például tegyük fel, hogy a nyelvi beállítás Case Insensitive, és magyar. Legyen egy oszlop neve cString. Ekkor miért ne hivatkozhatnánk *(mondjuk egy scriptben)* az oszlopra, mint cstring, hisz kis-nagybetû nem számít? Hivatkozhatunk, de ekkor jön a hibaüzenet, hogy nem ismer ilyen oszlopot. Miért? Mert „cs" egyenlô csé, míg cS egyenlô cées – legalábbis a magyar nyelvben. És a kiszolgálóban az összehasonlítások is átalakultak magyarrá…
Ezek a tulajdonságok nagyon úgy hangzottak, mint az SQL Server 7-nél megismertek. Mi a különbség? Amikor létrehozunk egy új adatbázist, akkor az a model adatbázis másolataként jön létre, azaz indirekt módon örökölni fogja azokat a nyelvi beállításokat, amelyeket a szerverpéldányra adtunk meg *(hacsak nem módosítottuk a model-t)*. Viszont ettôl – ellentétben a 7-es verzióval – eltérhetünk az adatbázis létrehozásakor, a COLLATE kulcsszó használatával. Azaz a globális beállításoktól függetlenül megadhatunk olyan nyelvet, amely alatt szeretnénk mûködtetni az adatbázisunkat.

Beállítási lehetôségek

A következôkben a LangTest nevû adatbázison keresztül bemutatom a különbözô nyelvi beállítási lehetôségeket az SQL Server 2000-ben.
Az SQL Server példány alapértelmezett nyelvi beállításának lekérdezése:

```
SELECT SERVERPROPERTY ('Collation')
SQL_Hungarian_CP1250_CI_AS
```

Az adatbázis nyelvi beállításának lekérdezése:

```
SELECT
  DATABASEPROPERTYEX ('LangTest', 'Collation')
SQL_Hungarian_CP1250_CI_AS
```

Az adatbázis nyelvi beállításának megadása vagy megváltoztatása:

```
CREATE illetve ALTER DATABASE LangTest
COLLATE Hungarian_CI_AI
```

Tábla létrehozása, többféle nyelvû oszloppal:

```
CREATE TABLE VegyesDuma
(
  AngolSzoveg varchar(500)
  COLLATE Latin1_General_CI_AS,
  MagyarSzoveg varchar(500)
  COLLATE Hungarian_CI_AS,
  NemetSzoveg varchar(500)
  COLLATE German_PhoneBook_CI_AS,
  --Ez olyan collation-ű lesz, mint az adatbázis
  DBDefaultSzoveg varchar(500)
)
```

A lehetséges nyelvi konstansok listája:

```
SELECT * FROM ::fn_helpcollations()

name                description
----                -----------
...
Hindi_CS_AS_KS_WS Hindi, case-sensitive, accent...
Hungarian_BIN     Hungarian, binary sort
...
```

Csak a magyarok altípusai:

```
SELECT name FROM ::fn_helpcollations()
  WHERE name LIKE 'Hungarian%'

Hungarian_BIN
Hungarian_CI_AI
Hungarian_CI_AI_WS
Hungarian_CI_AI_KS
...
```

Egy kis magyarázat a fenti konstansokhoz. Az elsô rész a nyelvet vagy nyelvcsaládot jelenti. Utána a Case Sensiti-

vity jelzése *(S=Sensitive, I=Insensitive)*. Az „A” mint Accent, teljesen hasonlóan. A K és a W pedig a Kana ill. Width érzékenységet jelöli. Azaz, ha nekem olyan magyar beállítás kell, ami nem érzékeny a kis-nagybetûre, de megkülönbözteti az ékezeteket *(ez a tipikus beállítás)*, akkor a Hungarian_CI_AS a nyerô választás. Valójában kétféle nyelvi konstans csoport is van, az egyik a Windows 2000 által is biztosítottakkal azonos, a másik rész pedig csak az SQL Server által biztosítottak. Ezeket legtöbbször csak a régi SQL Serverekrôl történô frissítés során használjuk, az ajánlott *(és sokkal bôvebb)* az elsô csoport.

Nyelvi csatározások

Mi van akkor, ha egy tábla különbözô oszlopaiban különbözô nyelveken írt szövegeket szeretnénk tárolni, az adott nyelvnek megfelelôen sorbarendezni, kezelni? Az egyik oszlopban egy kis magyar szöveg, a következôben német, satöbbi. Természetesen ennek semmi akadálya, köszönhetôen annak, hogy akár oszlopszinten megmondhatjuk a nyelvi beállításokat. Azonban ez jó kis kalamajkákhoz tud vezetni, amint valamilyen módon kapcsolatba kerülnek egymással.

Ha különbözô collation-û oszlopokat akarunk összehasonlítani *(<>=)*, akkor gondok lesznek, hisz melyiken értelmezett beállításokat, pl. kis-nagybetû érzékenységet vegye figyelembe az SQL Server? Ugyanez a kérdés jön elô különbözô nyelvû szövegek összefûzésénél *(+)* is. Magyart a héberrel összehasonlítani semmi értelme, de pl. jól jöhet az, hogy egy oszlopban tárolt szöveg ékezetre érzékeny beállítású, míg egy másik nem, és ezeket szeretnénk összehasonlítani valamely módon, például ékezetek nélkül. Ami még ennél is gyakoribb, hogy az összehasonlítások során hol érzékenynek kell lenni a kis-nagybetûre, hol nem. Ilyenkor jön jól a COLLATE kulcsszó, amellyel igazságot lehet tenni. Ám elôbb nézzük meg, hogy a különbözô helyekrôl jövô nyelvi beállításoknak mekkora a prioritása egymással szemben.

Ha egy beépített függvény szöveget ad vissza *(pl. USER_NAME)*, akkor annak a collation-je az aktuális adatbázisra *(amiben a felhasználó van)* alapértelmezett beállítás lesz. Ez a leggyengébb prioritású.

Adatbázisoszlopra történô hivatkozásnál, az oszlop adatbázisból örökölt vagy a tábla létrahozásakor megadott nyelvi beállítás lesz az alap.

Ha egy kifejezésben *(például két szöveg összehasonlítása)* explicit megadunk egy collation-t valamelyik szövegre, akkor általában annak lesz a legnagyobb prioritása.

Nézzük meg az elôbbieket egy példán keresztül! Készítsünk egy táblát, amelynek két oszlopa van, mindkettô magyar beállításokkal, de az egyik kis-nagybetû érzékeny, a másik nem.

```
CREATE TABLE Comp
(
  HCI varchar(50)
 COLLATE SQL_Hungarian_CP1250_CS_AS NOT NULL,
  HCS varchar(50)
  COLLATE SQL_Hungarian_CP1250_CI_AS NOT NULL
)
```

Szúrjunk be egy tesztsort, és próbáljuk meg összehasonlítani a két oszlopot!

```
INSERT Comp (HCI, HCS)
  VALUES ('Netacademia', 'NetAcademia')

--Teszt egyenlőségvizsgálat
SELECT
  CASE
    WHEN HCI = HCS THEN
    'Egyformák'
  ELSE
    'Különböznek'
  END,
  HCI,
  HCS
FROM
  Comp
```

Az összehasonlítás helyett egy csúnya hibaüzenetet kapunk:

```
Server: Msg 446, Level 16, State 9, Line 1
Cannot resolve collation conflict for equal to
operation.
```

Az a problémája az SQL Servernek, hogy össze akarunk hasonlítani két oszlopot, amelyeknek nem azonos a collation-je. Ez önmagában még nem volna probléma, csakhogy az egyenlôség mindkét oldalán oszlop-hivatkozás található, így a prioritások alapján nem lehet eldönteni az összehasonlítás kivitelezéséhez szükséges kis-nagybetû érzékenységet, mert a két oldal egyforma prioritású.

Mit lehet tenni? Egyrészt megmondhatnánk kiszolgálónak, hogy a jobboldali oszlop, amely egyébként kis-nagybetû érzékeny, ne legyen az. Így megszûnik a konfliktus oka, hisz a kifejezés mindkét fele azonos collation-û lesz:

```
...
  CASE
  WHEN HCI =
       HCS COLLATE SQL_Hungarian_CP1250_CI_AS
...
```

Az eredmény az elvárt lesz, azaz a két oszlop megegyezik, mert kis-nagybetûre nem érzékeny collation-t választottunk közös nevezônek:

```
            HCI          HCS
---------- ----------- -----------
Egyformák  Netacademia NetAcademia
```

Aki szereti a konfliktust, annak ajánlok egy másik megoldást. Melyik konstrukciónak is volt a legnagyobb prioritása? Hát az explicit COLLATE parancsnak. Mondjuk azt, hogy tudjuk, hogy a jobb oldali oszlop Case Sensitive, és ezt szeretnénk megerôsíteni egy COLLATE-el is. Ekkor ez ellentmondás lesz a HCI oszlop által dirigált Case Insensitivity-vel szemben, de hát gyôz az erôsebb:

```
...
  CASE
  WHEN HCI =
       HCS COLLATE SQL_Hungarian_CP1250_CS_AS
...

             HCI          HCS
----------- ----------- -----------
Különböznek Netacademia NetAcademia
```

A nyelvi beállítások miatti ütközéseknek, lehetséges konfliktusoknak most csak egy részét elemeztem ki. Részletes információt velük kapcsolatban a BOL-ban, a Collation Precedence címszó alatt találhat a Kedves Olvasó.

Zárszó

Az elôzô Tech.net számban kimaradt egy rész a Transact SQL sorozatból. Sok visszajelzést kaptam, amelyekben hiányolták az aktuális havi SQL Server evangéliumot. Köszönöm a dicsérô szavakat. Minden lelkes SQL Server rajongónak üzenem, hogy amíg lesz tinta a billentyûzetemben, addig lesz SQL sorozat is. A pozitív visszajelzések jót tesznek a töltôtollamnak, a kritikák pedig segítenek abban, hogy a sorozat még jobb legyen, és még inkább arról szóljon, ami segítik az Önök mindennapi munkáját.

Soczó Zsolt MCSE, MCSD, MCDBA
Zsolt.Soczo@netacademia.net