Zárolások az SQL 2000-ben

A mai, korszerű adatbázisok egyik legfontosabb jellemzője, hogy sokan használják egyidejűleg. 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 időtúllépésről, 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 következő 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 jelzőt a rangja elé. Ezzel együtt a lakcíme is megváltozott, amiről külön értesíti az egyik HR-es hölgyet.
Az emberi erőforrás 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 előtt ki vannak listázva Béla adatai, és nekiállnak módosítani a rekordot. Az első a lakcímet és a rangot, a második csak a rangot írja át. Megnyomják az „Elment" gombot, és tegyük fel, hogy az első hölgy a gyorsabb. Mi történik?
Két eset lehetséges. Egy butább adatbázis kezelő vagy egy rosszul megírt ügyfél alkalmazás esetén az első 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 elsőt, azaz a végleges rekordban nem lesz módosítva a lakcím, mert a második hölgy csak a rang mezőt módosította. A probléma nem az, hogy ez így megtörténhet, hanem az, hogy az ügyfél programok nem is szereznek róla tudomást, hogy módosításvesztés történt.
Az iménti helyzetben felvázolt helyzetet hívjuk az *elveszett módosítások problémájá*nak. Hogyan védekezik az ilyen helyzetek ellen egy okos adatbáziskezelő? Amikor egy ügyfél program lekér egy adott rekordot az adatbáziskezelőtől, 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 első ügyfél *optimista zárolás* felhasználásával kérte le a rekordot, akkor a hasonlóan eljáró további ügyfelek is megkapják a rekordot. Azonban módosítás visszaírási kísérlet esetén az adatbáziskezelő 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 megfelelő 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 első alkalmazás, ami módosítani akar egy rekordot lefoglalja azt, és a többiek addig nem is tudják lekérni a rekordot mindaddig, amíg az első fel nem oldja a zárolást. Ezt a stratégiát *pesszimista zárolásnak* hívjuk. Ez is egy elfogadható hozzáállás, ráadásul egyszerűbb implementálni a várakozást, mint lekezelni a sikertelen módosítást. (Gyakorlatilag nem kell tenni semmit, mert az adatbázist elérő metódus nem tér vissza addig, amíg a módosítandó rekord fel nem szabadul.)
Például egy helyfoglaló rendszernél csak ez a módszer tud helyesen működni, 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 helyzetről beszéltem, amikor a rekordokat kurzor segítségével kértük le, és a kapott recordset-en 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 ügyfél program (a Webalkalmazás) kapcsolata az adatbázissal, így az adatbázisnak még esélye sincs arra, hogy zárolással vagy Voodoo varázslással megóvjon minket a módosítások elvesztésétől.
Tegye a szívére a kezét minden Webalkalmazás fejlesztő! Gondolt már valaha erre a problémára? Vagy csak mechanikusan visszaírja a módosított eredményeket a forrás táblába? Vesszen a lassabb? Az ADO természetesen ilyen helyzetekre is biztosít megoldást, de használjuk ezeket? (A jövőben mindenképpen áldozunk egy-két cikket a témának.)

SQL Server tranzakciók

Szakadjuk el egy kicsit a kurzort használó ügyfél programoktól, 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 védendő 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 adat módosító tranzakció közepette nem lenne szerencsés engedni a többi tranzakciót, hogy olvassa az éppen módosítás alatt álló adatokat, pláne, 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 művelet 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 követhetőség kedvéért nem fordítottam le a zárolások nevét, és az egyszerűbb 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 adat módosító 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 adat módosítás befejeződik, és a tranzakciót így vagy úgy, de be nem fejezik. A legtöbb esetben ezzel az esettel kerülnek szembe az adatbázis fejlesztők és üzemeltetők, azaz, hogy egy hosszú ideig tartó adatmódosító tranzakció zárol egy bizonyos adatmennyiséget, így az egyéb adat olvasó vagy módosító tranzakcióknak várni kell a módosítás befejezéséig. Ezt sokan tévesen dead-lock-nak azonosítják, pedig ennek semmi köze nincs ahhoz. Egyszerűen csak egy hosszú idejű tranzakció blokkolja a többi tranzakció munkáját. Az SQL Server Enterprise Manager Management, Current Activity, Lock/Process ID alatt találhatjuk meg a szerveren a zárolásokat megjelenítő 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 bűnös tranzakciót. Aki nem szereti a grafikus felületeket, annak az `sp_lock` tárolt eljárást ajánlom a zárolások megfigyelésére.
Az `UPDATE` rendhagyó művelet a többi háromhoz képest, mert az első 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 ő kettős 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 előtt 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 megelőzésében nagyon fontos szerepe van az Update lock-nak, amiről a következő számban írok bővebben.
Schema Modification lock-ot az adatbázis szerkezetét módosító utasítások (például `ALTER TABLE`) helyeznek el a megfelelő objektumokon, hogy közben nehogy mások is megpróbálják ugyanazt módosítani.
A lekérdezések fordítása közben a szerver *Schema Stability* lock-al akadályozza meg a lekérdezésben szereplő 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.
Amennyiben a zárolás több mint 8 egybefüggő lapot érint, az SQL Server Extent lock-ot helyez el a lapcsoportra (8 lapot hívunk extent-nek).
Egyes esetekben, amikor olyan sok módosítás történik, hogy az szinte egy egész tábla tartalmát érinti, a szerver inkább zárolja az egész táblát, semmint egyedi extent-eket, ezzel a zárolások nyilvántartásához szükséges erőforrásokat spórolva.
Az SQL Server automatikusan választja ki, hogy mikor milyen finomságú zárolásra van szükség. A tranzakció által érintett sorok számától függően keres olyan szintű zárolást, ami még elég finom ahhoz, hogy ne korlátozza jelentősen 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 lapok vagy extent-ek, esetleg 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 szintű 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 sokkal 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 idejű lekérdezést futtatunk, ami múltbeli adatokkal foglalkozik, miközben záporoznak be a táblába a mai naphoz tartozó sorok. Lehet, hogy a lekérdezés akár a sorok első 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 lap szintű zárolást erőltetve 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 szűkíteni, az alkalmazásunk logikájától függően.

Hogyan befolyásolhatjuk az SQL Servert a zárolások finomságát illetően? A kérdésre a lock hint-ek adnak választ, a cikk utolsó részében.
A végére hagytam egy különleges zárolási típust, amely az előbbiekkel ellentétben nem fix méretű zárolást valósít meg. Ez az *index-tartományzárolás*. Bizonyos esetekben (`SERIALIZABLE` tranzakciók, lásd később) 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 index tartomány két végét lezárja *Key lock*-al, 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 működni, ha a tartományokat definiáló mezőre 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 ő is elhelyezheti a Shared lock-jait ugyanazokon a sorokon. Ha eközben egy harmadik résztvevő 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 ebből adódik e zárolás neve is.
Azaz abban az esetben, ha egy tranzakció szeretne valamilyen zárolást elhelyezni egy adathalmazon, az SQL Server ellenőrzi, hogy a már fennálló zárolások alapján kiadható-e a kért típusú zárolás. Ha igen, akkor megkapja, a zárolás feljegyzésre kerül, és a trónkövetelő tranzakció megkezdheti a munkáját. Amennyiben viszont olyan zárolást kért, ami logikailag nem összeegyeztethető a már meglévőkkel, akkor a zárolást kérő utasítást a szerver mindaddig felfüggeszti, amíg meg nem szűnnek az akadályozó zárolások. Az igényt természetesen feljegyzi, és a többi zárolás fokozatos „lehullása" alatt mindig ránéz, hátha már kiadható a kért zárolás. Miközben az igénylő vár a lock-jára, lehet, hogy más tranzakciók is jelentkeznek zárolási igénnyel, és azok között akár olyan is lehet, ami összeegyeztethető lenne a már fennálló zárolásokkal. Ilyenkor mit tegyen a szerver? Engedje őket, hogy elhelyezzék a saját zárolásaikat, vagy addig ne engedje őket szóhoz jutni, amíg a már régóta várakozó tranzakció meg nem kapja az áhított zárolását? Ha engedi őket, akkor azok lefuthatnak a várakozó előtt, ám előfordulhatna az, hogy a sok újabb és újabb kérő 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 meglévőkkel. A gyakorlatban ez azt jelentené, hogy egy módosító utasítás soha nem kapná meg az Exclusive lock-ját, ha az egymás után érkező olvasó (`SELECT`), Shared lock-okat elhelyező utasításokkal operáló tranzakciók időben átlapolják egymást. Nyilván ezt nem engedhetjük meg. Emiatt az SQL Server nem engedi zárolni a további kérőket, 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-al védett sorokat, de nem tehetik, mert a módosító utasítás vár az Exclusive lock-jára. Gyönyörű hosszú blokkolási láncok tudnak így kialakulni. Mit lehet tenni ellenük?
A legegyszerűbb védekezés, hogy a módosító tranzakciókat nagyon rövidre tervezzük. Nem szabad egy adat módosító 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, amelyekről hamarosan szólok.
Egy valamiről 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 tervezői 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 című 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 különböző finomságú zárolásokat? Ha van egy Exclusive lock egy soron, akkor rakható Shared lock ugyanarra a táblára? Ilyen kérdőjeles helyzet nagyon sok kialakul, hiszen minden tranzakció más finomságú zárolást használhat. Nézzünk erre egy példát.
Az első tranzakció Shared lock-al 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 szintű 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 szintű Shared lock-ot. Ha közben egy-egy tranzakció befejeződik, 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 szintű 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 kérő tranzakció igénye könnyen eldönthető, hisz elég megnézni, hogy van-e nem kompatíbilis Intent lock a táblán.
Ez az eljárás nem csak tábla szinten működik, hanem minden olyan szinten, amikor egy kisebb finomságú zárolást kér egy tranzakció. Így egy Exclusive sor lock-ot kérő tranzakció kap egy „valódi" Exclusive lock-ot a soron, és kap egy lap és tábla szintű 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 előfordulhat, 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 létező 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 működését. Természetesen egy adatbázisban nem alapozhatunk „többé-kevésbé" szabályokra, valamilyen egzakt módszer kell annak eldöntésére, hogy miközben az egyik tranzakció valamit működik egy táblán, a többi tranzakció ebből 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 előre 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 szereplő összes `SELECT` utasítás az adott elszigeteltségi szintnek megfelelően 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 megfelelően fognak működni. Ez azonban nem jelenti azt, hogy az előtte levő `SELECT`-ek által lefoglalt zárolások feloldódnának, csak azt, hogy az ezután kiadottak az új szintnek megfelelően fognak viselkedni. Igazából nem sok szituáció indokolja a szintek váltogatását egy tranzakció során, általában az elején beállítunk egy nekünk megfelelő szintet, és azt használjuk a tranzakció végéig. Lássuk hát a négy szintet!

1. READ UNCOMMITTED (dirty read)

Ezen a szinten a tranzakcióban szereplő 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, ami még nincsenek véglegesen lerögzítve az adatbázisba, azaz a módosító tranzakció végén még nem volt `COMMIT TRAN`, és lehet, hogy a következő 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 őt beszúró vagy módosító adatbázis véglegesítette a változtatását.
Azonban sokszor nem fontos az adatok hajszálra menő precizitása, de fontos, hogy a tranzakciónk ne blokkoljon más tranzakciókat a sok és hosszú idejű 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 nem 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-jű adatbázisra vonatkoznak, ami a vizsgált Northwind. Mit jelent ez a kimenet? Az első 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 megszűntek volna. Azt azonban könnyű megfigyelni a következő 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 rögzítődtek a táblába. Mit lát ebből 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 műveletek nem foglalkoznak még az Exclusive lock-okkal sem.
Hogy megnyugodjanak a kedélyek, görgessük vissza az előbbi 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 megfelelő sor, lap vagy tábla kiolvasása megtörtént, a zárolások feloldódnak. Amennyiben a `SELECT` halad előre 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. Ellenkező esetben visszalépnénk az előző 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 előző 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 első lekérdezést!
Mit látunk? Semmit. A lekérdezés csak fut, csak fut... Mivel a másik tranzakció adatmódosítása Exclusive lock-okat helyezett el a tábla sorain (sőt 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 első tranzakció is befejezi a futását, és kiadja az eredeti, módosítás előtti 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 első 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 különböző időpillanatban. Lehet, hogy más tranzakció megváltoztatja az általunk kiolvasandó sorokat a két kiolvasás között, ezt nevezzük nem megismételhető olvasásnak (non-repeatable read). Az is előfordulhat, hogy beszúrnak olyan sorokat a két SELECT közötti időben, amelyek megjelennek a második SELECT eredményhalmazában. Ezeket a megjelent sorokat hívják fantomoknak (phantoms). A következő 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 megismételve 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 köszönhetően. 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 befelyezéséig.
Ezen a szinten már nagyon erősen érezhető 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 őket.
Ha megnézzük, milyen zárolások keletkeznek ezen a szinten, akkor a következőt 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 kérdése lapokra vagy az egész táblára, mert úgysem fog sikerülni, hisz az adott „nagy" tartományokon belül vannak olyan sorok, amelyek Shared lock-al védettek.
Miért van két sor és lap zárolás, amikor a lekérdezés kimenete csak egy sort tartalmaz? Láthatjuk, hogy különböző indexekhez (IndId oszlop) tartoznak a zárolások. Az Employees táblán három index is van, ezek közül kettőt használt a lekérdezés. A LastName-re szűrtünk, ehhez a LastName oszlopra definiált Nonclustered index-et használta a szerver (ez könnyen ellenőrizhető a végrehajtási terv megtekintésével is). Miután megtalálta a LastName index táblában a megfelelő sorokat (jelen esetben 1 sor), a Nonclustered index, mint sor azonosító segítségével kiolvassa a megfelelő sor tartalmát. Ahhoz, hogy biztosítsa a zárolást bármelyik index-et használó tranzakció elől, 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 előbbi lekérdezést, aminek feltétel része a következő volt:

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

E szint logikájának megfelelően a szervernek le kell zárolnia az összes olyan lehetséges index irányokat, amelyeken keresztül B betűvel kezdődő nevű 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 betűvel kezdődő sorok közötti rész), így oda nem lehet új sorokat beszúrni.
A szintek tárgyalásánál nem szóltam az `sp_lock` kimenetéből az utolsó oszlopról. Abban látható, hogy a zárolást megkapta-e a kérő, vagy csak vár rá. Az összes példámban a mező értéke `GRANT` volt, azaz a kérő 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 működéstől, 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 első 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 kiterjedésű zárolást, mint a sor szintű. Hasonlóan a `PAGLOCK`, `TABLOCK` lap illetve tábla szintű zárolás használatára kéri a szervert.
Példa:

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

Az előbbi módosítók a zárolás finomságát állították. A következők a zárolás típusát szabályozzák.
Az `UPDLOCK` segítségével a `SELECT` az alapértelmezetten használt Shared lock helyett Update lock-ot helyez el az olvasott táblán. Ennek előnye, 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 működő 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 nem csak 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 nevű izolációs szint párja.
Az utolsó hint egy kicsit más, mint az előzőek. A `READPAST` azt mondja egy `SELECT` utasításnak, hogy egyszerűen ugorja át azokat a sorokat, amelyeket más tranzakció zárolt, és olvassa fel a nem zárolt sorokat. Ez csak `READ COMMITTED` elszigeteltségi szintű tranzakciókban működik, és csak a sor szintű zárolásokat tudja átlépni. Egy adatbázis elmélettel

foglalkozó embernek ettől 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 működik, 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 ő milyen objektumon végez zárolást (a nevét tudja, de a belső struktúrájáról semmit nem tud), csak van neki egy táblázata, amely alapján eldönti, hogy az ütköző zárolás kérések esetén továbbengedheti-e az igénylőt, 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 egyszerű. Az sp_getapplock tárolt eljárás meghívásával kérünk egy általunk megálmodott zárolási típust, egyedi néven. Elindítjuk a védendő, 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 egyszerűen megoldhatjuk:

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

EXEC VédendőTároltEljárás

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

Az sp_getapplock első 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 zároláson csak ő futtathassa a VédendőTároltEljárás-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 különböző 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 közbenső eljárást, akkor a 'Session' helyett 'Transaction'-t kell írni, és a három eljáráshívást tranzakcióba (BEGIN TRAN, COMMIT TRAN) kell foglalni. Ekkor a zárolás tranzakció szintű 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 jellegű zárolási sémákat is, amelyek megfelelően 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 kérdőjeles részekhez előkerült a Books Online is), már megtette az első lépést abban az irányba, ami a professzionális adatbázis tervezés felé vezet. Az adatbázis zárolási eljárásának ismerete nélkül tranzakciókat és adatbázisokat tervezni vakrepülés, amely előbb-utóbb egy sziklafalon végződik.
A következő cikkbe szorult át a dead-lock-ok elmélete és gyakorlata, amely azonban csak a zárolások ismeretében érthető meg. Visszavárom Önöket a halálos ölelések szigetén, a következő számban!

Soczó Zsolt MCSE, MCSD, MCDBA
Zsolt.Soczo@netacademia.net
Net Academia Kft.