MySQL 5 – rég várt fejlesztések

Megnézzük, hogy állnak a „legnépszerűbb nyílt forrású adatbázis-kezelő” programmal a fejlesztők.

Lassan 10 éve, hogy a *MySQL* töretlen pályája elkezdődött. A siker talán annak volt köszönhető (és ez így van a mai napig), hogy a pehelysúlyú kategóriát célozták meg a fejlesztők. Ennek a döntésnek a legfőbb előnye, hogy a *MySQL* egyszerű és gyors, könnyen megtanulható a használata, ideális a mindennapos igények kielégítésére. Hátrányként jelentkezik azonban, hogy a szolgáltatások köre szűkebb, mint a vetélytársaké. Ez ellen úgy próbáltak védekezni, hogy szép fokozatosan bővítették a lehetőséget, közben ügyelve rá, hogy ha nincs rájuk szükség, akkor épp olyan egyszerűen lehessen használni az adatbázis-kezelőt, mint annak előtte.
Az adatbázis-kezelés témakörében már jó ideje jelen lévő tárolt eljárások megjelenésével a *MySQL* készítői régi adósságukat törlesztették. Cikkünkben ezt, és egyéb, a témához kapcsolódó fejlesztéseket vizsgálunk meg közelebbről.

Helyzetkép

A cikk írásának pillanatában az 5.0.2-alpha fejlesztői változat érhető el „legfrissebb kiadás” címén a *MySQL* honlapján. Az egyes lehetőségek megvalósítása és beépítése a programba folyamatosan történik, minden kiadás tartalmaz egy-két újítást, s mellette rengeteg hibajavítást. A végleges változat tehát nem csak annyival fog többet tudni, hogy alaposan le lesz tesztelve, de jónéhány, jelenleg nem szereplő szolgáltatással is bővülni fog. Nekünk azonban jelenleg meg kell elégednünk a fenti változatszámú fejlesztői kiadással, de aggodalomra semmi ok, hisz ez is megfelelően körvonalazza számunkra, hogy egy esetleges következő alkalmazásunknál milyen lehetőségeket vehetünk igénybe, ha a *MySQL* adatbázis-kezelőt választjuk.

Nézzük az elsőt: nézetek

Az 5-ös változatban már lehetőségünk nyílik ***nézetek (view)*** készítésére. Egy ilyen nézet nem más, mint egy *SQL* lekérdezés által reprezentált adathalmaz, amelyet úgy érhetünk el, mintha az egy önálló tábla volna. Jó hír, hogy a nézetek mindjárt irhatók is, tehát lehetőségünk van a ***rekordok módosítására (update)***, illetve új sor ***beszúrására (insert)***, bár ez utóbbi bonyolultabb nézetek esetén nehézségekbe ütközhet, de ilyenre egyébként is csak a legritkább esetben vetemedünk.
A nézetek készítése viszonylag szabványosan történik, de van néhány megkötés. A nézet adathalmazát meghatározó lekérdezés nem tartalmazhat származtatott táblákat, nem hivatkozhat felhasználói változókra, nem hivatkozhat átmeneti táblákra, és ami az egyik legfontosabb: nem hozhatunk létre triggereket (lásd később) a nézet soraira.

Származtatott táblák, allekérdezések

A származtatott táblákat a 4.1-es változatban vezették be a készítők. Ez nem más, mint egy lekérdezés eredménye, amelyet felhasználunk egy másik, külső lekérdezésben mint táblát. Legfőképp abban különbözik a nézettől, hogy ez csak az adott lekérdezés idejéig létezik.
Példa:

```
select nev from (select azonosito,nev,cim from
➥szemelyek where azonosito>100) t2;
```

Ennek a lekérdezésnek persze a világon semmi értelme, de a származtatott tábla fogalma azért érzékelhető.
A fentihez hasonló dolog az ***allekérdezés (subquery)*** fogalma, amely a `WHERE` ágban elhelyezett lekérdezésre vonatkozik. Ez általában csak egyetlen mezővel tér vissza, s az alábbi formában szoktuk használni:

```
select nev from szemelyek where nev not in
➥(select nev from tiltott_szemelyek) tsz;
```

A példa egy negatív ***illesztést (join)*** hajt végre a két táblán, persze nem a leggyorsabb módon.

Tárolt rutinok

Egy tárolt eljárás olyan *SQL* nyelven írt parancsok összessége, amelyet az adatbázis-kiszolgálón rögzítünk, s utána a programozásban használatos módon meghívhatjuk. Ekkor elvégzi azokat az *SQL* utasításokat, amelyeket az eljárás törzsében meghatároztunk. Néhány bővítménynek köszönhetően tehát gyakorlatilag programozhatunk az adatbázis-kezelő berkein belül.
A megoldás egyik nagy előnye, hogy az adatbázissal kapcsolatos logika valóban az adatbázis szintjére kerülhet, nem teszi nehezen érthetővé a programkódot. A másik, talán ennél is fontosabb előny, hogy az itt elvégzett műveletekben szereplő rekordok nem hagyják el az adatbázis-kiszolgálót, hanem ott helyben történik meg a feldolgozásuk. Ennek következtében nem csak a hálózati terhelés csökken, de

Felhasználói változók

A *felhasználó változók (user variables)* olyan *MySQL* elemek, amelyekben a programozásban megszokott módon mindenféle értékeket tárolhatunk, s később hivatkozhatunk rájuk. Az ilyen változók jellegüket tekintve globálisak az adott kapcsolaton belül. Ez azt jelenti, hogy az adott ügyfélkapcsolaton belül bármelyik adatbázis használata közben elérhető, viszont senki más nem láthatja az általunk beállított változókat. Segítségükkel a különböző utasítások között rögzíthetünk állapotokat, teremthetünk kapcsolatot. Egy változó értékét a `SET @változónév = érték` paranccsal állíthatjuk be, mintha csak `UPDATE` művelettel volna dolgunk, az értékek kiolvasását pedig a `SELECT @változónév` paranccsal tehetjük meg, vagy simán a nevével hivatkozhatunk rá – például egy `WHERE` ágban.

mivel nincs késleltetés, maga a művelet is sokkal gyorsabb, mint a hagyományos esetben, arról már nem is beszélve, hogy az adatbázis-kiszolgáló ki tudja használni a saját maga által nyújtott gyorsítási lehetőségeket (például indexek).
A *MySQL* a tárolt eljárások kezelése során az *SQL:2003* szabványt követi, hasonlóan az *IBM* által fejlesztett *DB2* adatbázis-kezelőhöz.
Ennek megfelelően van *tárolt eljárásunk (stored procedure)* és van *tárolt függvényünk (stored function)*, amelyeket a leírás közös néven *tárolt rutinként (stored routine)* emleget. A kettő között a lényegi különbség az, hogy a függvénynek van valódi visszatérési értéke. Na de nézzük ezt meg részletesebben.

Az eljárások

Olyan visszatérési érték nélküli eljárásokról van szó, amelyek tetszőleges számú paramétert fogadva azoknak megfelelően végrehajtanak egy utasításhalmazt. Eddig nincs is benne semmi különös, a dolog ott kezd érdekessé válni, amikor megnézzük ezeket a paramétereket. Minden paraméter alapvetően háromféle lehet: `IN`, `OUT`, és `INOUT` (KI, BE és KI-BE) típusú, amelyek azt határozzák meg, hogy az adott paraméter bemeneti paraméter-e, vagy kimeneti, esetleg kétéltű. A bemeneti paraméterek a programozásban megszokott paraméterek, ezeket használjuk a függvényen belül, a kimeneti típusú paraméterek pedig visszatérési értéket képeznek. Egy ilyen kimeneti típusú paraméter csakis *felhasználói változó (user variable)* lehet, míg a bemeneti paraméter lehet konstans érték is (pl. egy szám).
A jelleg és a paraméter név megadása után következhet a típus megadása, ami jelen pillanatban a *MySQL* által használt típusok (oszlop típusok) lehetnek.
A tárolt eljárások a `CREATE PROCEDURE` parancs segítségével deklarálhatók, és ha egynél több utasítást tartalmaz, akkor `BEGIN` és `END` kulcsszavak közé kell ágyazni az eljárás törzsét. Elkészülte után a `Call <tárolt eljárás neve(paraméterek)>` módon hívhatók.
További hasznos érdekesség az eljárásoknál, hogy használhatjuk benne a jól megszokott `SELECT`-es lekérdezéseket, amelyek a hagyományos esetben visszaadódnak, mint az eljárás eredményei, tehát az eljárás futtatása ilyen esetben egyenértékű egy lekérdezés futtatásával. Fontos azonban megjegyezni, hogy ha több `SELECT` is lefut egy tárolt eljáráson belül, azok eredményei külön keresési eredményként adódnak vissza, tehát jó, ha a kliens tudja kezelni az összetett lekérdezési eredményeket.
Lássunk egy egyszerű példát az eljárások alkalmazására:

```
CREATE PROCEDURE negyzet(IN szam INT,OUT negyzete
➥ INT)
BEGIN
        SELECT szam*szam into negyzete;
END

Az eljárást az alábbi módon hívhatjuk:
mysql> CALL negyzet(4,@a);
Query OK, 0 rows affected (0.07 sec)
mysql> SELECT @a;
+------+
| @a   |
+------+
| 16   |
+------+
1 row in set (0.00 sec)
```

A példában a `SELECT INTO` szerkezetet alkalmaztuk, amely azt tudja, hogy a `SELECT` által visszatért értéket elhelyezi az általunk megadott változóba.

Függvények

A függvények rendelkeznek valódi visszatérési értékkel, s segítségükkel olyan számítási műveleteket végezhetünk, amelyek nem kapcsolódnak az adatbázishoz, illetve az ott tárolt adatokhoz, ugyanis egy függvényen belül nem használhatjuk a `SELECT`, `INSERT`, `UPDATE` és a hasonló műveleteket, tehát itt csak a kapott paraméterekkel dolgozhatunk, amelyek itt mindig bejövő értékeket takarnak.
Hasonlóan az eljárásokhoz a függvények a `CREATE FUNCTION` parancs segítségével deklarálhatók, és ha egynél több utasítást tartalmaz, akkor `BEGIN` és `END` kulcsszavak közé kell ágyazni az függvény törzsét.
Minden függvénynek van visszatérési-érték típusa, amelyet a deklaráció során adunk meg. Hasonlóan az eljárásokhoz, mind a függvény paraméterei, mind a visszatérési érték típusa *MySQL* által használt típus kell legyen.
Lássunk egy példát a függvények alkalmazására is:

```
CREATE FUNCTION funcdemo(szam int) RETURNS INT
BEGIN
RETURN szam*szam;
END
```

A függvényt az alábbi módon hívhatjuk:

```
mysql> select funcdemo(4);
+-------------+
| funcdemo(4) |
+-------------+
|          16 |
+-------------+
1 row in set (0.00 sec)
```

Megjegyzés: Mind az eljárások, mind a függvények esetében a törzs több BEGIN-END blokkból állhat, amelyek mindegyike felcímkézhető.

Vezérlési szerkezetek

A tárolt rutinok önmagukban még nem jelentenének nagy áttörést, azonban az elérhető vezérlési szerkezeteknek köszönhetően valóban programozhatunk *SQL* nyelven. Van IF, van CASE, REPEAT és WHILE ciklus, nincs FOR ciklus, van viszont egy érdekes LOOP nevű szerkezet, amely azt tudja, hogy a hurok kezdete és vége közötti részt ismételgeti, egészen addig, amíg a hurkon belül a megfelelő LEAVE utasítással a hurok elhagyására késztetjük a végrehajtást. Ezeken kívül létezik még egy ITERATE parancs, amivel a különböző ciklusokon belül kezdeményezhetjük a ciklus ismételt végrehajtását. (Az egyes szerkezetek szintaktikája megtekinthető a *MySQL* honlapján.)
Az itt felsorakoztatott lista teljes, és bár igen kevésnek tűnik, az a tapasztalatom, hogy a gyakorlatban elegendő, és az összes felmerülő probléma elvégezhető az elemek kombinálásával.

Változók, elemek

Csupán a vezérlési szerkezetekkel még nem sokra mennénk, a programíráshoz változók is kellenek, és a felhasználói változók nem feltétlenül alkalmasak erre a célra.
A már emlegetett BEGIN-END blokkon belül azonban lehetőségünk van különböző *MySQL* elemek és változók deklarálására. Nézzük meg közelebbről ezeket az elemeket:

- ***Feltételek (CONDITION)***: Segítségükkel hibakódokhoz rendelhetünk neveket, amely nevekre aztán később hivatkozhatunk, amikor le akarjuk kezelni a feltételek által reprezentált hibát
- ***Hibakezelők (HANDLER)***: Ezeket az elemeket ***kivételkezelésre (exception)*** használhatjuk, ahol a kivételt a *MySQL* kiszolgáló dobja egy hibakód formájában, mi pedig egy ilyen hibakezelő segítségével rögzíthetjük, hogy az adott hibakód felmerülése esetén milyen műveletet kell végrehajtani.
- ***Változók (VARIABLES)***: Ezek a hagyományos értelemben vett lokális változók, amelyek *MySQL* által használt típusúak lehetnek és rendelkezhetnek kezdőértékkel, de nem azonosak a felhasználói változókkal.

Mindhárom elemtípust a DECLARE kulcsszóval kell bevezetni, a pontos szintaktika a *MySQL* leírásban található.

Kurzorok

A kurzor nem más, mint egy lekérdezés eredményének olyan átmeneti tárolási lehetősége, ahonnan később akár rekordonként is visszahozhatjuk az abban tárolt adatokat. *MySQL*-ben minden kurzorhoz tartozik egy lekérdezés, amelyet a deklaráció során kell megadni. A kurzort a későbbiekben egy változó reprezentálja, amelyet megnyithatunk, értékeket olvashatunk ki belőle, majd lezárhatunk. Természetesen ezt is a DECLARE kulcsszóval vezethetjük be, és csak a tárolt rutin határait jelző BEGIN-END blokkon belül érvényes.
Van egy olyan szabály, miszerint nem mindegy, hogy milyen sorrendben deklaráljuk a fentebb emlegetett elemeket: hibakezelő után nem deklarálhatunk kurzort, és a változók deklarációjának mind a hibakezelők, mind a kurzorok deklarációját meg kell előzniük. Ha ebből deriválunk, az alábbi sorrendet kapjuk: változók, kurzorok, hibakezelők.
Kicsit térjünk most vissza a tárolt eljárásokhoz, s lássuk, hogy a fentieknek mi köze az egészhez. A kurzorok nélkülözhetetlenek az eljárásokban lekérdezett eredményhalmazok ciklikus bejárása érdekében, itt ugyanis nincs FOR SELECT (amely végigmegy egy lekérdezés eredményhalmazán), mint más alkalmazások esetében. A dolgot úgy kell áthidalni, hogy készítünk egy kurzort, amely tartalmazza a lekérdezés eredményeit, majd ezen kurzor minden elemén végighaladva járhatjuk be az eredményeket. A ciklus akkor ér véget, amikor elérjük a kurzor végét. Ezt – jobb híján – egy hibakezelő létrehozásával kell megoldani, amely bebillent egy jelzőbitet, amit a REPEAT-UNTIL ciklusban figyelünk. Még mielőtt megijednénk, hogy ez milyen bonyolult (bár más *SQL* megoldásokhoz képest határozottan az), lássunk egy példát a fentire:

```
create procedure curdemo()
BEGIN
DECLARE a int;
DECLARE b char(255);
DECLARE done INT DEFAULT 0;
DECLARE cur1 CURSOR FOR SELECT id,nev from
➥automarkak;
DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET
➥done = 1;
OPEN cur1;
REPEAT
    FETCH cur1 INTO a, b;
    IF NOT done THEN
                SELECT a as id, b as nev;
    END IF;
  UNTIL done END REPEAT;
END
```

Ez a példa semmi hasznosat nem csinál, csupán kiír néhány autómárkát és nevet, sajnos mindet külön lekérdezés eredményeképp. A fentebb leírt módszert viszont jól szemlélteti. A done nevű változó jelzi, ha elfogytak az eredményhalmazból a rekordok, mivel a 02000-s SQLSTATE akkor következik be, ha az eredményhalmaz végére értünk.

Kis összegzés

Bár még nem értünk a cikk végére, hadd húzzak itt egy vonalat, és írjam le, ami a tesztelés során felhalmozódott bennem. Több helyen is le van írva a *MySQL* dokumentációjában, hogy a rendszer még fejlesztés alatt áll, de azt hiszem, hogy ha nem lenne leírva, erre akkor is hamar rájönnénk. Igen szegényes az elérhető lehetőségek köre. A legfőbb nehézség az, hogy nem tudunk úgynevezett leválogatást készíteni, amely valamilyen szempontok szerint kiszűrné egy lekérdezés eredményét, és csak a kritériumoknak megfelelő rekordokat adná vissza, ugyanis az eljárásnak nincs valódi visszatérési értéke, a SELECT által visszaadott értékek viszont soronként különböző lekérdezési eredményként adják vissza a kívánt rekordokat. Nem lehet típust definiálni, amivel áthidalhatnánk a visszatérési értékek problémáját, bár azzal itt nem is sokra mennénk, mivel a függvények nem nyúlhatnak az

adatbázisban tárolt adatokhoz (de attól még hiányoznak az összetett változótípusok). Ezen túl körülményes, hogy csak a kurzorokon keresztül lehet hozzányúlni ciklikusan az eredményekhez, a kurzorok viszont rendkívül buták. Nem lehet bármilyen eredményt hozzáadni, nem lehet később hozzáfűzni eredményeket, nem lehet benne pozicionálni, és még sorolhatnám. Szóval van még mit fejleszteni.

Vissza a témához: Triggerek

A *triggerek* olyan speciális tárolt eljárások, amelyek valamilyen esemény (`INSERT`, `UPDATE`, `DELETE`) esemény során hívódnak meg automatikusan. A trigger egy adott táblához van kötve, és az arra a táblára vonatkozó, előre megadott művelet során hajtódik végre.
Általában adatbázisba írás előtti ellenőrzésre, írás utáni számított érték kiszámítására, törlés előtti adatok bizonyos részeinek rögzítésére, stb. használhatjuk.
Megadható, hogy az adott esemény előtt (`BEFORE`), vagy után (`AFTER`) hajtódjon végre, egyébként egy szokványos tárolt eljárásról van szó, vagyis inkább függvényről... Eljárás, mert nincs visszatérési értéke, függvény, mert rengeteg olyan megkötés van, amit előzőleg a függvényeknél tapasztalhattunk: nem lehet másik tárolt eljárást, vagy függvényt meghívni, nem lehet tranzakciót kezdeményezni, jóváhagyni vagy visszavonni, de ami a legfontosabb: nem hivatkozhatunk közvetlenül más táblákra, de még arra a táblára sem, amihez a triggert rendeltük. Egyedül az aktuálisan érintett rekord értékeit kaphatjuk meg, vagy írhatjuk felül az alábbiak szerint: `INSERT` esetén a beillesztendő rekord oszlopai elérhetők a `NEW.oszlopnév` hivatkozással. Ebben az esetben az oszlopnév írható, azaz a `SET NEW.oszlopnév=érték` művelet lefut, ha megvan a megfelelő jogosultságunk. `DELETE` esetében a törlendő rekord oszlopai érhetők el az `OLD.oszlopnév` szintakszissal, de csak olvasható módon. `UPDATE` művelet esetén a felülírandó rekord értékeit az `OLD`, az új rekord értékeit pedig a `NEW` hivatkozáson keresztül lehet elérni (előbbi csak olvasható, az utóbbi írható).
Ezen kívül van még egy olyan – egyébként természetes – megkötés, hogy egy eseményhez egy adott időben (tehát például `BEFORE INSERT`) csak egyetlen trigger tartozhat.
Ha a fenti kivételeket leszámítjuk, akkor ugyanúgy használhatjuk, mint egy függvényt, élnek a vezérlési szerkezetek, vannak változóink, de nem láthatunk ki a függvényből.
Nézzünk egy példát trigger használatára:

```
CREATE TRIGGER ins_szorzo BEFORE INSERT ON termekek
    -> FOR EACH ROW SET NEW.fogyar = NEW.nagykerar
➥* 1.2;
```

Ez a trigger beszúrás előtt kiszámítja a fogyasztói árat a nagykereskedelmi ár felszorzásával, majd maga az `INSERT` művelet csak ez után fut le, tehát az érték bekerül a táblába – anélkül, hogy a „felhasználói” oldalról bármit csináltunk volna.
És még egy példa a *MySQL* dokumentációból, amely egy `UPDATE` eseményre levágja a megadott intervallumból kilógó értékeket. Íme:

```
mysql> delimiter //
mysql> CREATE TRIGGER upd_check BEFORE UPDATE ON
➥account
    -> FOR EACH ROW
    -> BEGIN
    ->     IF NEW.amount < 0 THEN
    ->         SET NEW.amount = 0;
    ->     ELSEIF NEW.amount > 100 THEN
    ->         SET NEW.amount = 100;
    ->     END IF;
    -> END//
mysql> delimiter ;
```

És ha már ide érkeztünk: hasznos újítás a `delimiter` parancs, amely arra szolgál, hogy megadjuk vele, hogy az `sql` parancs bevitele során (parancssoros kliens estén) milyen karakter zárja le magát a parancsot. Mivel az alapértelmezett határoló karakter a pontosvessző, ami egyben a tárolt rutinok nyelvének utasításait is lezárja, kénytelenek vagyunk átdefiniálni ezt addig, amíg begépeljük (bemásoljuk) a rutint. Jelen esetben a `//` karaktereket használjuk határolónak, majd mikor végeztünk, visszaállítjuk az alapértelmezett pontosvesszőre.
Sajnos a jelenlegi rengeteg korlátozás miatt még néhány kevésbé bonyolult dologra sem tudjuk felhasználni, és a dolgozni sem egyszerű vele.

Összegzés

Bár a fejlesztők törekvése helyes és tiszteletreméltó, még igencsak gyerekcipőben jár ez a dolog. A program készítői minden lehetséges ponton feltüntették, hogy a legtöbb lehetőség jelenleg is bővítés alatt áll, és ezt tudomásul véve is azt kell mondjam, hogy komoly dolgokra jelenleg (és még pár hónapig ez így is marad) nem alkalmas. Hiába a pehelysúlyú kategória zászlós hajója a *MySQL*, ezen projekt esetében ez nem számít, ugyanis a cikkben tárgyalt témakörök nem igazán sorolhatók ide, de ha figyelembe vesszük, hogy a tárolt rutinok kezelésénél soha nem fog a program a tökéletes sokoldalúságra törekedni, a tudása még akkor is gyenge. Hozzáteszem, az irány jól el van találva, csak még nem haladt az alkalmazás kellő számú lépést ezen az egyébként rögös úton. Nem beszélve az előbb már érintett problémáról, a pehelysúlyú megoldások alkalmazásáról egy inkább nehézsúlyú területen. Magam is kíváncsi vagyok, hogy a készítők hogyan fogják ezt a feladatot megoldani. Azt javaslom, várjunk még… érdemes próbálgatni, feltérképezni a jelenlegi fejlesztői változatot, és bár ezekre nem szoktunk alkalmazásokat építeni, feltételezem, hogy az első stabil változtok sem fogják tartalmazni a teljes fegyverarzenált, így ha valaki *MySQL*-ben szeretne magasröptű adattárolási megoldásokat tárolt rutinokkal kezelni, az talán várjon egy kicsit, s előbb néhány közepes erősségű probléma megoldására törekedjen. Majdnem biztos vagyok benne, hogy idővel ezen a területen is mélyen a szívünkbe lopja magát a *MySQL*.

Komáromi Zoltán

KAPCSOLÓDÓ CÍMEK

A MySQL honlapja: ➲ http://www.mysql.com
Magyar tükör: ➲ http://mysql.sote.hu