

Malé veľké databázy III / 1.časť

Hovorí sa, že tam, kde niečo končí, iné zase začína. Oficiálny školský rok sa blíži k úspešnému koncu, ale my začíname. Preto vás vítam na našej vysokej škole databáz. Ako som naznačoval v minulej časti, budeme sa v tejto tretej sérii venovať špecifickým problémom v oblasti databáz. Jasné, že bez nich by naše projektíky a aplikácie fungovali, ale ak chceme dosiahnuť vyššiu kvalitu, rýchlosť alebo spoľahlivosť, bez indexov, zamykania, či archivácie dát to nedosiahneme. Dnes sa budeme venovať kľúčom a indexom.

Vráťme sa na chvíľu k našim začiatkom a zopakujme si, čo vieme o kľúčoch a indexoch vo všeobecnej teórii databáz.

Predstavme si akúsi tabuľku, napríklad telefónny zoznam. Pre zjednodušenie taký kancelársky typ, písaný na malých kartičkách, ktoré bývajú vložené do kruhového otočného stojanu, ako som to už mnohokrát videl v amerických filmoch. (Aby sme si to hneď zo začiatku nekomplikovali, budeme predpokladať, že v tomto zozname budeme mať iba jedno meno na každé písmeno abecedy, teda na písmeno „T“ bude iba pán Takáč a už nikto iný na „T“.) Nech sú na tej kartičke uvedené tieto údaje: **Poradové_Číslo_Kartičky**, **Priezvisko**, **Meno**, **Bydlisko** a **Telefónne_Číslo**. A ako vznikajú také kartičky? No predsa postupne, tak ako vznikajú obchodné či osobné kontakty. Pre názornosť: hneď prvý deň si vytvoríme kartičku s číslom 1, kde si zapíšeme údaje o pánovi Krátkom, bývajúcom v Bratislave s telefónnym číslom 75689625. Založíme do stojana. Druhý deň stretneme slečnu Blaníkovú, a keďže sa nám veľmi páči, rýchlo zavedieme kartičku č.2 s jej telefónnym číslom a adresou a šup s ním do stojana. A takto postupujeme odo dňa ku dňu a kartotéka sa naplňa a naplňa a jej obsah môže vyzeráť asi takto:

Poradove_Číslo_Karty	Priezvisko	Meno	Bydlisko	Telefonne_Číslo
1	Krátky	Ján	Bratislava	02/75689625
2	Blaníková	Alena	Žilina	041/3698524
3	Žabka	Pavol	Martin	043/3217538
.....				
244	Takáč	Peter	Banská Bystrica	048/7896542
245	Stehlíková	Karolína	Brezno	047/9513578
246	Václavík	Klement	Trenčín	032/6567896
.....				

Skúsme teraz náš imaginárny telefónny zoznam popísať už „databázisticky“:

Vidíme, že je to akási tabuľka o n- záznamoch a 5 stĺpcoch. Záznamy rastú vzostupne podľa

Poradové_Číslo_Kartičky. Môžeme povedať, že zrovna tento stĺpec je akýmsi identifikátorom celej tabuľky.

Keďže čísla zapisujeme pekne vzostupne, nemôže sa stať, že by sa v tomto stĺci nachádzali dve rovnaké hodnoty a tak môžeme povedať, že tento stĺpec je **jednoznačný** a je ideálny ako primárny kľúč tabuľky.

Ak sa pozrieme na položku **Priezvisko**, ktorá asi bude dominantná pri vyhľadávaní telefónneho čísla, zistíme, že bohužiaľ nie je vôbec zotriedená podľa abecedy, lebo sme kartičky vyplňali tak, ako sme získavali kontakty, teda pekne poporiadku. Ak budeme chcieť vyhľadať telefónne číslo paní Stehlíkovej, máme dve možnosti: buď prelistujeme celú kartotéku pekne od začiatku až narazíme na meno Stehlíková, alebo...alebo budeme mať akúsi pomocnú tabuľku, ktorá bude zotriedená podľa abecedy a v nej bude zaznamenané, že priezvisko Stehlíková sa v hlavnej tabuľke nachádza na kartičke s číslom 245. V tejto pomocnej tabuľke už vyhľadáme veľmi rýchlo.

Apriori vieme, že „S“ bude po písmene „R“ a pred písmenom „T“, takže všetky predchádzajúce riadky môžeme preskočiť. Pre nás je podstatný údaj v druhom stĺpci takejto pomocnej tabuľky, ktorý sa odkazuje na príslušné miesto v tabuľke hlavnej. Tejto pomocnej tabuľke sa hovorí **index**. S takýmito pomocnými indexovými tabuľkami či kartotékami sa môžeme hlavne stretnúť v knižniciach. Sú to tie malé kartičky, uložené v šuplíkoch, kde môžeme vyhľadávať podľa názvu - názvový index, alebo podľa autora - autorský index. Je to omnoho pohodlnejšie, ako prechádzať celé police s knihami a hľadať tú správnu knihu.

Vráťme sa k našemu telefónnemu zoznamu:
A takto vyzerá papierová forma tabuľky **index**:

Priezvisko	Karta
Blaníková	2
Krátky	1
.....	
Stehlíková	245
Takáč	244
.....	
Václavík	246
Žabka	3

V elektronickej podobe to vyzerá veľmi obdobne, kde k hlavnej tabuľke je nadefinovaný index na konkrétny stĺpec hlavnej tabuľky. Ak by sme si to chceli graficky predstaviť, naša hlavná tabuľka **ZOZNAM** a index na stĺpec **Priezvisko**, nazvaný napríklad **idx_priez**, a ich vzájomné vzťahy by vyzerali asi takto (obr.č.1-1):



Index je datová štruktúra, uložená v samostatnom súbore, ktorá urýchľuje vyhľadávanie a triedenie v hlavnej tabuľke **ZOZNAM** podľa stĺpca **Priezvisko**, teda v tomto prípade podľa abecedy.

Funguje ako určitý abecedný oddeľovač uložený v súbore. Umožňuje nájsť tú časť abecedy, ktorú práve hľadáme, v našom prípade písmeno „S“ a tak preskočiť všetky predchádzajúce písmená. Vyhľadávanie je tak omnoho rýchlejšie a program nestráca čas prehládávaním záznamov, ktoré ho nezaujímajú.

Indexy sú výborná vec, ale majú aj svoje nevýhody. Príliš mnoho indexov môže mať opačný efekt.

V predchádzajúcom príklade nás index okamžite prenesie do oboru záznamu začínajúcim na písmeno „S“. Ale ako by asi index vyzeral, kedy sme neindexovali iba prvé písmeno, ale priamo celé mená? Index by tak obsahoval kvantá oddeľovačov - prakticky toľko, koľko záznamov je v tabuľke uložených. Vyhľadávanie by tak bolo skôr pomalšie. Preto by sme mali vo svojich tabuľkách udržiavať rozumný počet indexov a ich rozsahov.

Ďalším problémom je, že pridávanie záznamov do indexovaných tabuliek je omnoho pomalšie, ako do neindexovaných. Je to logické - zatiaľ čo v neindexovanej tabuľke sa nový záznam jednoducho „prilepí“ na koniec tabuľky, v indexovanej musí okrem pridania ešte prehľadať všetky oddeľovače v indexe a zaradiť nový oddeľovač na správne miesto.

Stále ale platí, že potom je ale vyhľadávanie podstatne rýchlejšie. Sami teda musíme rozhodnúť, čo je pre nás výhodnejšie. Neskôr si povieme o jednoduchšej finte, ako urobiť, aby „bola koza celá a vlk sýty“.

Zapamätajte si! Indexy urýchľujú prístup k dátam prostredníctvom SQL príkazu typu *SELECT*. Znižujú však výkon príkazov *INSERT*, *UPDATE* a *DELETE*.

Aké polia indexovať

Ak sme sa rozhodli používať indexy, musíme sa najprv rozhodnúť, ktoré polia tabuľky indexovať. Rozhodnutie nebýva vždy jednoduché. Spravidla chceme indexovať tie polia, ktoré sa používajú najčastejšie. Existuje také pravidlo - sú to tie polia, ktoré sa v tej-ktorej aplikácii uvádzajú v klauzuli **WHERE**.

V SQL príkaze

SELECT * FROM ZOZNAM WHERE Priezvisko = 'Stehlíková'

sa k indexácii ponúka stĺpec **Priezvisko**.

Môže však nastať táto teoretická situácia: Sme mimo pracoviska a tak naša sekretárka preberá telefóny a odkazy. Keď sa vrátíme, nájdeme na stole poznámku, že máme volať tel. číslo 048/7896542. Ale ani za svet si nevieme spomenúť, komu to číslo patrí! No a vypytovať sa „Haló, kam som sa dovolal?“ je nevhodné a tak chceme nájsť v našej tabuľke vlastníka tohoto čísla. A aby to šlo rýchlejšie, najlepšie bude mať aj index na pole

Telefonne_Cislo.

V jednej tabuľke môže byť definovaných viac kľúčov. Aj vyššie spomínaný príklad má najmenej dva kľúče - **idx_priez** a **idx_tel**. Každá tabuľka môže mať najviac 16 indexov, ale také množstvo by sme nemali používať. Ak sa však tomu nemôžeme vyhnúť, mali by sme znovu prehodnotiť celý návrh databáze, aby sme sa vyhli problémom. Niekoľko indexov na každú tabuľku však nie je neobyčajný jav.

Spravidla sa osvedčuje indexácia polí obsahujúcich jedinečné dáta. Preto sa prednostne indexujú *primárne kľúče*. To je jeden z dôvodov, prečo si užívatelia zamieňajú pojmy *kľúč* a *index*. Kľúče zjednodušujú definíciu štruktúry databáze, zatiaľ čo indexy zvyšujú jej výkon. Konkrétnejší rozdiel medzi kľúčmi a indexami si teraz vysvetlíme.

Rozdiel medzi kľúčom a indexom

Vo všeobecnej teórii databáz je **kľúč** štruktúrny odkaz v databáze, ktorý definuje reláciu (vzťah) medzi dvomi tabuľkami. Kedykoľvek budeme chcieť prepojiť tabuľky navzájom, použijeme kľúče z jednotlivých tabuliek. Najvhodnejší je na to samozrejme **primárny kľúč**. Spomeňme si na definíciu primárneho kľúča - musí byť čo najkratší, musí byť jedinečný (teda nemôže obsahovať dve rovnaké hodnoty) a nesmie obsahovať prázdnu hodnotu (NULL).

Keď sa pozrieme na našu tabuľku ZOZNAM, vidíme, že ideálnym predstaviteľom primárneho kľúča je stĺpec **Poradove_Cislo_Karty**. Ak budeme chcieť využiť túto tabuľku na spojenie s inou tabuľkou, napr. na evidenciu telefónnych hovorov s kontaktnými osobami (nech sa táto tabuľka nazýva **HOVORY**), budeme využívať práve tento kľúč:

ID_hovoru	Datum	Cas_trvania	ID_volaneho
1	23022002	12:32	1
2	24032002	2:45	246
....		
1569	27042002	7:08	245

Ak zadáme príkaz

**SELECT Hovory.*, Zoznam.Priezvisko, Zoznam.Bydlisko FROM Hovory, Zoznam
WHERE Hovory.ID_volaneho = Zoznam.Poradove_Cislo_Karty**

dostaneme celkom pekný výpis už s konkrétnymi priezviskami a bydliskom volaných osôb.

A tu sme práve využili primárny kľúč.

Voľba primárneho kľúča je pri návrhu databáze veľmi dôležitá. Primárny kľúč je základným stavebným kameňom ukladania dát a je to v skutočnosti on, čo umožňuje spájanie tabuliek a vlastnú realizáciu koncepcie relačných databáz. Voľba primárneho kľúča musí byť dobre premyslená, pretože iba on dokáže jednoznačne identifikovať každú záznam v tabuľke.

Predstavme si, že v položke **Priezvisko** sa môžu nachádzať dva rovnaké záznamy. Je to reálne, veď koľko existuje ľudí z rovnakým priezviskom. Z toho vyplýva, že položka **Priezvisko** ako primárny kľúč neprichádza do úvahy. Môžeme ale tento stĺpec využiť ako index - tak, ako sme si to vysvetlili vyššie.

Primárny kľúč je v MySQL automaticky indexovaný, takže sa rozdiel medzi kľúčmi a indexami čiastočne stiera. Pre jednoduchosť teda budeme považovať v MySQL slová index a kľúč za synonymá (= slová rovnakého významu).

Typy indexov

Aké teda typy indexov (=kľúčov) v MySQL poznáme?

Sú to:

- Ø primárny kľúč - *primary key*
- Ø jedinečný kľúč tiež nazývaný jedinečný index - *unique*
- Ø jednoduchý index - *index*

Stĺpec tabuľky, nad ktorým chceme vytvoriť jednoduchý index, môže obsahovať aj položky s rovnakými hodnotami. Vzorovým príkladom môže byť stĺpec **Priezvisko** v tabuľke **ZOZNAM**.

Jedinečný kľúč (unique index)

Ako už vyplýva zo samotného názvu, jedinečný index nedovoľuje v indexovanom poli existenciu duplicitných položiek. Ak sa pokúsime do poľa s jedinečným indexom vložiť duplicitný záznam, t.j. taký, ktorý tam už existuje, systém to nedovolí a vyhlási chybu.

Aký je teda rozdiel medzi jedinečným (unique) a primárnym (primary) kľúčom?

Každá tabuľka môže mať iba jeden primárny kľúč, ale viac unikátnych kľúčov. A ešte jedna výhoda pri použití unique kľúčov - pri odstránení primárneho kľúča z tabuľky sa novým primárnym kľúčom stane automaticky unikátny kľúč.

Vytváranie indexov

Indexy - kľúče môžeme vytvárať niekoľkými spôsobmi, a to pri vytváraní novej tabuľky alebo na už vytvorenú (a naplnenú) tabuľku.

Definícia indexu pri vytváraní tabuľky

Samotné vytváranie tabuliek je tak, ako všetko ostatné pomerne jednoduché. Aj tu existuje niekoľko rôznych spôsobov.

Všeobecný zápis je:

```
CREATE TABLE meno_tabulky
(
.....
INDEX meno_indexu (položka1,položkaN),
UNIQUE meno_indexu (položka1,položkaN),
PRIMARY KEY meno_indexu (položka1,položkaN),
.....
)
```

Mená indexov volíme vhodne tak, aby popisovali položku, na ktorú boli nastavené. Ja najradšej používam prefix **idx_** a pripojím k tomu (skrátенý) názov indexovanej položky, napr. **idx_priez** a **idx_tel**.

V uvedenom zápise si všimnime, že index môže byť zložený aj z viacerých položiek, nielen z jednej. Vtedy zapíšeme zoznam indexovaných polí do zátvoriek a oddelíme ich od seba čiarkou.

Predstavme si vytvorenie tabuľky **ZOZNAM**, kde primárnym kľúčom bude stĺpec **Poradove_Cislo_Karty**, a indexy budú na stĺpoch **Priezvisko** a **Telefonne_Cislo**:

```
CREATE TABLE Zoznam
(
Poradove_Cislo_Karty INT Auto_Increment PRIMARY KEY NOT NULL,
Priezvisko VARCHAR(30),
Meno VARCHAR(15),
Bydlisko VARCHAR(20),
Telefonne_Cislo VARCHAR(12),
INDEX idx_Priez (Priezvisko),
```

```
INDEX idx_tel (Telefonne_Cislo)
)
```

Vytvorenie indexov na už existujúcu tabuľku

Ak sme vytvorili tabuľku **ZOZNAM** len s primárnym kľúčom, ale bez indexov, môžeme tieto dodefinovať takto:

```
CREATE INDEX meno_indexu ON meno_tabulky (zoznam_indexovaných_polí)
```

napr.:

```
CREATE INDEX idx_priez ON Zoznam (Priezvisko).
```

Ak by sme chceli dodefinovať k niektorej tabuľke jedinečný - unikátny index, syntaktický zápis bude vyzerat' takto:

```
CREATE UNIQUE meno_indexu ON meno_tabulky (zoznam_indexovaných_polí)
```

Poznámka: Takýmto spôsobom nie je možné definovať *primary key*.

Ešte vhodnejším spôsobom na vytvorenie indexu, unikátneho indexu ale aj primárneho kľúča je použitie klauzuly **ALTER TABLE**. Syntaktické zápisy sú tieto:

```
ALTER TABLE meno_tabulky ADD INDEX meno_indexu (zoznam_indexovaných_polí)
ALTER TABLE meno_tabulky ADD UNIQUE meno_indexu (zoznam_indexovaných_polí)
ALTER TABLE meno_tabulky ADD PRIMARY KEY (zoznam_indexovaných_polí)
```

Odstraňovanie indexov

Za určitých okolností (povieme si akých) je vhodné index odstrániť. Na to slúži táto skupina identických SQL príkazov:

```
DROP INDEX meno_indexu ON meno_tabulky
```

alebo

```
ALTER TABLE meno_tabulky DROP INDEX meno_indexu
```

alebo obdobne

```
ALTER TABLE meno_tabulky DROP PRIMARY KEY
```

Kedže indexy sú samostatné štruktúry uložené v samostatnom súbore na disku s príponou *.MYI*, mazať index nie je nebezpečné. Žiadne dáta z hlavnej tabuľky sa nestratia. Naopak, musíme mať na pamäti, že ak zmažeme stĺpec, pridáme aj o jeho index.

Zisťovanie údajov o indexoch

Ak potrebujem zistiť údaje o tabuľke a jej indexoch, môžeme použiť dva príkazy v monitore *mysql*.

Na popis štruktúry tabuľky použijeme (zobáček > naznačuje zadávanie príkazu v monitore *mysql*):

```
> describe zoznam;
```

a dostaneme výsledok, aký je na obr. 1-2:

Field	Type	Null	Key	Default	Extra
Poradove_Cislo_Karty	int(3)		PRI		auto_increment
Priezvisko	varchar(30)	YES	MUL	0	
Meno	varchar(15)	YES		0	
Bydlisko	varchar(20)	YES		0	
Telefonne_Cislo	varchar(12)	YES	MUL	0	

Všimnime si, že v stĺpci *Key* sú tri zápisy:

- PRI u položky **Poradove_Cislo_Karty** značí, že sa jedná o primárny kľúč - PRIMARY KEY
- MUL u položiek **Priezvisko** a **Telefonne_Cislo** značí, že k príslušnému poľu je pripojený index, ktorý nie je jedinečný

Pomocou nasledujúceho príkazu zistíme podrobnejšie informácie:

> **show index from zoznam;**

Výsledok je na obr.č.1-3:

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality
zoznam	0	PRIMARY	1	Poradove_Cislo_Karty	A	0
zoznam	1	idx_priez	1	Priezvisko	A	
zoznam	1	idx_tel	1	Telefonne_Cislo	A	

V stĺpci *Key_name* vidíme už skutočné mená indexov **idx_priez** a **idx_tel**.

Pri prezeraní tejto tabuľky nás najviac zaujímajú tieto stĺpce:

Non_unique značí, či takto indexovaná položka môže obsahovať duplicitné záznamy. 0 (nula) znamená, že nemôže. A je to pravda, lebo *Primary key* nemôže obsahovať duplicitu.

Collation značí, ako bude index triedený. „A“ značí vzostupne, „NULL“ značí netriedený.

Rôznorodosť indexov

Už vieme, že sa indexy skladajú z jednej alebo viacerých položiek.

Preto indexy rozdeľujeme na niekoľko druhov:

- Ø jednopoložkové indexy
- Ø viacpoložkové indexy
- Ø čiastočné indexy
- Ø zložené čiastočné indexy

Jednopoložkový index

Ak nastavujeme index v tabuľke iba na jednu položku, hovoríme o jednopoložkovom indexe. V tomto prípade uvedená položka dostatočne identifikuje záznam a je to asi najčastejší prípad použitia indexu.

Potom v zápise definície indexu je ako parameter iba táto položka, napr.:

ALTER TABLE Zoznam ADD INDEX idx_priez (Priezvisko)

Viacpoložkový index

V prípadoch, kedy nie je možné zaistiť jednoznačnosť záznamu pomocou jednej položky tabuľky, môžeme použiť dva či viac stĺpcov. Vhodný kľúč môže vzniknúť kombináciou (*Priezvisko*, *Meno*) alebo dokonca (*Priezvisko*, *Meno*, *Bydlisko*). Správna voľba závisí od aplikácie. Vzorový zápis potom bude:

ALTER TABLE Zoznam ADD INDEX idx_priez (Priezvisko, Meno, Bydlisko)

Čiastočný index

Už sme si hovorili, že ak je index veľmi rozsiahly, je jeho súbor veľmi veľký, ba dokonca môže byť väčší ako dátový súbor. Aj vyhľadávanie pomocou veľmi rozsiahleho indexu je nákladné na prostriedky systému. Príkladom môže byť vyššie spomínaný index na položku **Priezvisko**. Ak by sme mali v našej tabuľke viac mien, ktoré začínajú na to isté písmeno (napr. Stehlíková a Salaba) a použili by sme index z odstavca o jednopoložkovom indexe, boli by v indexe definované obidve mená, teda Odkaz na Stehlíková aj na Salabu. V praxi však stačí indexovať len na niekoľko prvých znakov položky. Povedzme, že nám v tomto prípade stačí indexovať len na prvé tri znaky z položky Priezvisko. Zápis vytvorenia indexu bude potom vyzeráť takto:

```
ALTER TABLE Zoznam ADD INDEX idx_priez (Priezvisko(3))
```

Takto vytvorený index bude pomerne malý a skutočne efektívny.

Zložený čiastočný index

Je kombináciou viacpoložkového a čiastočného indexu. Ak znova vystačíme s prvými tromi znakmi z každej položky, zápis bude:

```
ALTER TABLE Zoznam ADD INDEX idx_priez (Priezvisko(3), Meno(3), Bydlisko(3))
```

Je samozrejmé, že počet prvých znakov (tzv. predpona) môže byť rôzny pre každú položku.

Indexové voľby

Kedy je vhodné použiť index? A aký typ indexu? Existujú určité zaužívané pravidlá, ktoré vedú k úspechu.

Kedy je vhodné použiť index:

- Ø *Klauzula WHERE.* Ak niektorú položku používame veľmi často v klauzulke *WHERE*, nastavenie indexu na túto položku podstatne zrýchli výkon príkazu *SELECT*. Jedná sa niekedy o niekoľko tisíc-násobné zrýchlenie!
- Ø *Klauzula ORDER BY a GROUP BY.* Triedenie záznamov pomerne značne zaťažuje celý systém. Avšak indexy automaticky triedia záznamy v abecednom poradí, takže by sme ich mali použiť aj na položky, ktoré sa často používajú v klauzulách *ORDER BY* a *GROUP BY*.
- Ø *MIN() a MAX().* Pri vyhľadávaní najmenších a najväčších hodnôt sú indexy ideálnym pomocníkom.
- Ø *Spojovanie tabuliek.* Význam systému relačných databáz spočíva v schopnosti rýchlo vyhľadať a spojiť informácie, ktoré sú uložené v samostatných tabuľkách. Doporučuje sa, aby bola indexovaná väčšina položiek, ktoré sa na spojení tabuliek podieľajú.

Kedy by sa indexy použiť nemali:

- Ø *Tabuľky, do ktorých sa často zapisuje.* Pri každom zásahu do tabuľky, bez ohľadu na to, či použijeme príkaz *INSERT*, *UPDATE* alebo *DELETE*, je potrebné zmeny uložiť nielen do hlavnej tabuľky, ale aj do všetkých indexov a teda aj do ich súborov. V tomto prípade má existencia mnohých indexov za následok značné zaťaženie systému.
- Ø *Výberové príkazy - SELECT, ktorých výsledok obsahuje veľmi veľa záznamov.* Ak očakávame, že výsledok selektu obsahuje značné množstvo záznamov, nastáva pri použití indexu určité zdržanie, lebo systém musí pristupovať do dvoch súborov - dátového a indexového. Vtedy je efektnejšie, aby systém radšej tabuľku prehľadával rad po rade.
- Ø *Malé tabuľky.* Výhody indexov spoznáme iba pri mnohých záznamoch, nie pri desiatich či päťdesiatich.
- Ø *MySQL nedovoľuje vytvoriť index na polia, ktoré obsahujú prázdne hodnoty.*

Poznámka: Doporučuje sa, aby primárny kľúč obsahovala každá tabuľka v databáze, v ktorej chceme jednoznačne identifikovať každý záznam.

Kompromisy

Čo v takom prípade, že mám tabuľku, do ktorej sa často a veľa zapisuje, ale zároveň sa v nej často vyhľadáva? Popíšem náš príklad z praxe:

Máme tabuľku údajov o materiáli, v ktorej sa hlavne vyhľadáva príkazom *SELECT* počas celého dňa, lebo jej obsah je prístupný na intranete. Keďže sa nejedná o kritické informácie, ktoré by vyžadovali aktualizáciu “on-line”, môžeme si dovoliť raz za čas, spravidla v noci alebo po pracovnej dobe, doplniť nové údaje.

Vtedy urobíme toto:

- Ø Odstránime indexy príkazom *DROP INDEX*. Ako vieme, žiadne údaje sa nestratia.
- Ø Pridáme nové záznamy príkazom *INSERT*. Keďže indexy neexistujú, záznamy sa pridávajú rýchlo na koniec tabuľky.
- Ø Nastavíme príslušné indexy príkazom *ALTER TABLE ADD INDEX*. Takto sa vytvoria indexy na naplnených dátach - hovoríme tomu, že sa dáta zindexovali.

Nové prehľadávanie príkazom *SELECT* je opätovne veľmi rýchle, lebo posledná indexácia vždy zasiahne aj posledne pridané záznamy.

Ešte existuje jeden špeciálny typ indexu - tzv. **FULLTEXT** index. Ako už jeho názov naznačuje, slúži na fulltextové vyhľadávanie v tabuľkách. Že neviete, čo to je? Tak práve toto je asi najmocnejší index v súčasných databázach, ale o tom si povieme neskôr.

Malé veľké databázy III / 2.časť

V minulej časti sme si povedali dôležité veci o kľúčoch a indexoch. Teraz vieme, že správne postavenie indexov dokáže podstatne zrýchliť vyhľadávanie dát. Bohužiaľ, klasické indexy sa dali použiť len na jednoznačné dáta. Čo v takom prípade, ak chceme vyhľadávať text v rôznych stĺpcoch, keď presne nevieme, kde sa hľadaný reťazec nachádza alebo aký je presne jeho zápis? Na to sa v moderných databázových strojoch používa tzv. full-textové vyhľadávanie.

Full-text index

Full-textové vyhľadávanie môžeme preložiť ako „plnotextové vyhľadávanie“, teda vyhľadávanie v celom texte. Aby sme mohli zrealizovať full-textové vyhľadávanie, musíme v konkrétnej tabuľke vytvoriť full-textový index. Takýto index a vyhľadávanie podporuje MySQL od verzie 3.23.23.

Full-text index sa označuje **FULLTEXT** a môže byť vytvorený na stĺpce typu *VARCHAR* a *TEXT*.

Jeho vytvorenie je podobné ako u klasických indexov, teda použitím v príkaze **CREATE TABLE** alebo priamo príkazom **CREATE INDEX** či **ALTER TABLE**. Ak si spomenieme na zásady, ktoré sme si vysvetľovali minule, môže byť použitie *CREATE INDEX* alebo *ALTER TABLE* omnoho efektnejšie pri rozsiahlych databázach, ako vkladanie záznamov do prázdnej tabuľky s už nadefinovaným indexom. Ako vždy, správne rozhodnutie závisí od použitia aplikácie.

MATCH a AGAINST

Pri zavedení full-textového indexovania a vyhľadávania sa objavujú aj nové kľúčové slová **MATCH** a **AGAINST**. *MATCH* je zároveň aj funkciou. Jej úlohu si vysvetlíme neskôr.

MATCH môžeme voľne preložiť ako „porovnanie“.

AGAINST môžeme v tomto význame preložiť ako „s“ alebo „proti“.

Význam týchto spojení si vysvetlíme neskôr na konkrétnom príklade.

Vzorový príklad

Vráťme do našej virtuálnej knižnice a majme takúto jednoduchú tabuľku s názvom **KNIHY** o každej knihe:

id
nazov
popis

Nech obsahuje dva indexy - jeden primárny index na položku **id** a druhý full-textový index na položky **nazov** a **popis**. Je zrejmé, že tento druhý index bude tzv. *zložený index*.

SQL zápis vytvorenia takejto tabuľky potom bude:

```
CREATE TABLE knihy
(
    id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,
    nazov VARCHAR(50),
    popis TEXT,
    FULLTEXT (nazov,popis)
)
```

Pre ilustráciu príkladu si naplníme takuľku **KNIHY** týmito dátami:

id	nazov	popis
1	MySQL Tutorial	DBMS stands for DataBase Management ...
2	How To Use MySQL Efficiently	After you went through a ...
3	Optimising MySQL	In this tutorial we will show how to ...
4	1001 MySQL Trick	1. Never run mysqld as root. 2. Normalise ...
5	MySQL vs. YourSQL	In the following database comparison we ...
6	MySQL Security	When configured properly, MySQL could be ...

Zápis SQL príkazu bude:

INSERT INTO knihy VALUES

```
(0,'MySQL Tutorial', 'DBMS stands for DataBase Management ...'),
(0,'How To Use MySQL Efficiently', 'After you went through a ...'),
(0,'Optimising MySQL', 'In this tutorial we will show how to ...'),
(0,'1001 MySQL Trick', '1. Never run mysqld as root. 2. Normalise ...'),
(0,'MySQL vs. YourSQL', 'In the following database comparison we ...'),
(0,'MySQL Security', 'When configured properly, MySQL could be ...')
```

Je len samozrejmé, že nula na začiatku každého riadku značí automatické pridanie čísla v zmysle autoinkrementácie.

Podme si teraz vyskúšať full-textové vyhľadávanie:

Nájdime tie záznamy, kde sa niekde v stĺpcoch *nazov* alebo *popis* nachádza text *database*.

SQL príkaz pre túto požiadavku bude:

SELECT * from knihy WHERE MATCH (nazov,popis) AGAINST ('database')

Ako by sme v našom jazyku popísali tento príkaz? Môžeme to skúsiť aj takto: “Vypíš všetko z tabuľky *KNIHY*, kde v stĺpcoch *nazov* alebo *popis* nájdeš porovnanie so slovom ‘database’”.

Funkcia **MATCH** porovná dopyt v prirodzenom jazyku (alebo logickom - povieme si neskôr) s tvarom za kľúčovým slovom **AGAINST**. Táto funkcia je *case-insensitive*, teda nezáleží na veľkosti písmen.

Potom výsledok dopytu SQL, napr. v okne monitora mysql, je podobný výpisu č.III-2-1:

```
+-----+-----+-----+
| id | nazov | popis |
+-----+-----+-----+
| 5 | MySQL vs. YourSQL | In the following database comparison we ... |
| 1 | MySQL Tutorial | DBMS stands for DataBase Management ... |
+-----+-----+-----+
2 rows in set (0.17 sec)
```

Vidíme výpis dvoch riadkov, ktoré skutočne obsahujú slovo *database*.

Relevance (slov. relevancia)

Pre každý záznam v tabuľke sa vracia tzv. **relevance** - to je akýsi stupeň podobnosti medzi textom v zázname a v SQL príkaze. *Relevance* je nezáporné reálne číslo s plávajúcou čiarkou. Nulová relevancia značí žiadnu podobnosť. Relevanciu vypočíta SQL server na základe počtu slov v zázname, počtu jedinečných slov v tomto zázname, celkového počtu slov v sade a počtu záznamov, ktoré obsahujú časť slova.

Ukážme si, ako zistíme relevanciu slova *Tutorial*:

Zadáme SQL príkaz:

SELECT id, MATCH nazov,popis AGAINST ('Tutorial') from knihy

My vieme, že slovo Tutorial sa nachádza v 1. a 3. riadku. V týchto riadkoch funkcia MATCH vypočíta relevanciu podľa vyššie popísaného postupu. Je zrejmé, že číslo relevancie bude pre 1.riadok iné, ako pre 3.riadok. Na ostatných riadkoch bude hodnota rovná nule, lebo tam sa slovo nenachádza.

Výsledok vidíme na výpise č.III-2-2:

```
mysql> select id,MATCH(nazov,popis AGAINST ('Tutorial')) from knihy;
```

id	MATCH(nazov,popis AGAINST ('Tutorial'))
1	0.64840710366884
2	0
3	0.66266459031789
4	0
5	0
6	0

```
6 rows in set (0.22 sec)
```

MySQL používa veľmi jednoduchý parser (= syntaktický analyzátor) na analýzu slov. „SLOVO“ je v tomto prípade postupnosť písmen, číslíc a znakov apostrofu a podtržítka. Každé slovo, ktoré je zapísané v zozname „stopword list“ alebo má menej ako 3 znaky, je ignorované. **Stopword** je slovo, ktoré sa pri fulltextovom vyhľadávaní nezohľadňuje a vyhľadávanie sa ukončí a nevráti sa žiadny záznam.

Každé platné slovo je *vážené* podľa jeho významu. Teda slovo, ktoré sa nachádza vo viacerých záznamoch, má menšiu váhu ako slovo, ktorého výskyt je vzácny. *Váha* slov sa použije pri výpočte relevancie.

Zadajme teraz tento príkaz:

```
SELECT * from knihy where MATCH (nazov, popis) AGAINST ('MySQL')
```

Podľa všetkého predpokladáme, že by sa mali vypísať všetky záznamy, lebo v každom z nich sa toto slovo nachádza. Ale ouha, pozrime na výpis č.III-2-3:

```
mysql> select * from knihy where MATCH (nazov, popis) AGAINST ('MySQL');
Empty set (0.22 sec)
```

Nevrátili sa žiadne záznamy! Prečo?

Slovo „MySQL“ sa nachádza vo viac ako v polovici všetkých záznamov a tak je ako také v tomto prípade prezentované ako *stopword*, teda nemá žiadnu sémantickú hodnotu.

Je to vlastne požadované správanie sa SQL stroja, pretože si nevieme predstaviť, keby sa vrátil každý druhý záznam z napr. 1 GB tabuľky! Na podobnom princípe fungujú aj veľké internetové vyhľadávače.

Logické full-text vyhľadávanie

Od verzie 4.0.1 MySQL môže uskutočňovať aj *boolean fulltext* vyhľadávanie použitím **IN BOOLEAN MODE** parametra. Jeho úlohou je ovplyvňovať výsledok príkazu SELECT podľa určitých logických operátorov.

Použijeme SQL príkaz:

```
SELECT id, nazov from knihy WHERE MATCH (nazov, popis)
AGAINST ('+MySQL -YourSQL' IN BOOLEAN MODE)
```

Čo znamená?

Vypíš (nájd) stĺpec *id* a *nazov* z tabuľky *KNIHY*, kde je možné v stĺpcoch *nazov* alebo *popis* nájsť slovo *MySQL* (znamienko +), ale nesmie sa tam nachádzať slovo *YourSQL* (znamienko -).

Výsledok je na výpise č.III-2-4:

```
mysql> select id, nazov from knihy where MATCH (nazov,popis)
-> AGAINST ('+MySQL -YourSQL' IN BOOLEAN MODE);
```

id	nazov
1	MySQL Tutorial
2	How To Use MySQL Efficiently
3	Optimising MySQL
4	1001 MySQL Trick
6	MySQL Security

5 rows in set (0.28 sec)

Vidíme, že sa vrátili všetky záznamy obsahujúce slovo *MySQL* a neobsahujúce slovo *YourSQL*. Záznam číslo 5 túto podmienku nespĺňa a preto bol z výsledku vyradený.

Aj keď sa slovo *MySQL* nachádza vo viac ako 50% záznamov, toto obmedzenie sa tu nevyužilo, lebo slovo *YourSQL* túto obmedzujúcu podmienku 50% nespĺňa!

Operátory boolean fulltext vyhľadávania

V zápise za kľúčovým slovom **AGAINST** je možno používať tieto logické operátory:

- “+” Plus pred slovom znamená, že toto slovo sa musí nachádzať vo vrátených záznamoch
- “-” Mínus pred slovom znamená, že toto slovo sa nesmie nachádzať vo vrátených záznamoch
- “<” Tieto dva operátory sú používané na zväčšovanie alebo zmenšovanie relevačnej hodnoty slova. Pozri príklad nižšie
- “()” Zátvorky sú použité - ako obvykle - na združovanie výrazov
- “~” “Tilda” operátor je negátor. Používa sa pri označovaní rušivých slov. Záznam, ktorý obsahuje takto označené slovo má menšiu hodnotu ako ostatné, ale nie je vylúčený z vyhľadávania ako pri mínus operátore
- “*” Zrezanie slova - poznáme ako wildcards - divoké znaky

A tu je pár príkladov:

- “auto motor” nájde záznamy, ktoré obsahujú najmenej jedno z týchto slov
- “+auto +motor” nájde záznamy, ktoré obsahujú obidve slová
- “+auto motor” nájde slovo *auto*, ale hodnota relevancie je vyššia, ak záznam obsahuje aj slovo *motor*
- “+auto -motor” nájde záznamy, ktoré obsahujú slovo *auto*, ale neobsahujú *motor*
- “+auto +(>motor <brzda)” záznam môže obsahovať túto kombináciu: *auto a motor*, alebo *auto a brzda*, ale *auto motor* má väčšiu hodnotu relevancie ako *auto brzda*
- “auto*” možnosti sú: autobus, automat, autor....

Obmedzenia pri fulltextovom vyhľadávaní

Ako všetko, aj fulltextové vyhľadávanie má určité obmedzenia:

- všetky parametre u funkcie **MATCH** musia byť stĺpce z tej istej tabuľky okrem použitia *IN BOOLEAN MODE*
- zoznam stĺpcov medzi **MATCH** a **AGAINST** musia byť definované ako **FULLTEXT** index, okrem použitia *IN BOOLEAN MODE*
- argument v **AGAINST** musí byť konštantný reťazec

Vylad'ovanie fulltextového vyhľadávania

Nanešťastie, fulltextové vyhľadávanie má zatiaľ iba niekoľko nastavovacích parametrov. Vylad'ovanie je možné počas prekladu zo zdrojových kódov MySQL.

Tu je niekoľko možných nastavení:

- minimálna dĺžka slova pre indexáciu je definovaná v premennej *ft_min_word_lenght*
- zoznam stopwordov je definovaný v súbore *myisam/ft_static.c*
- prah 50% obmedzenia je možné vypnúť v súbore *ftdefs.h* zamenením položky

```
#define GWS_IN_USE GWS_PROB
```

na

```
#define GWS_IN_USE GWS_FREQ
```

a rekompileovaním MySQL
- ak by sme potrebovali zmeniť operátory použité pri logickom vyhľadávaní, nájdeme ich definované v premennej *ft_boolean_syntax* v súbore *ft_static.c*

Čo sa chystá do budúcnosti

V najbližších verziách programu MySQL sa v oblasti fulltextového vyhľadávania očakávajú tieto vylepšenia:

- všetky operátory s **FULLTEXT** indexom budú rýchlejšie
- zavedie sa tzv. frázové vyhľadávanie a príbuznosť operátorov
- logické (boolean) vyhľadávanie bude môcť fungovať aj bez **FULLTEXT** indexu
- podpora „vždy indexovaných“ slov. To môžu byť reťazce, ktoré užívateľ bude chcieť spracovať ako slová, napr. C++, AS/400, TCP/IP a iné
- podpora fulltextového vyhľadávania v *MERGE* tabuľkách
- podpora pre mnohobajtové znakové sady
- možnosť tvorby zoznamu *stopword listu* v závislosti od jazyka dát
- možnosť tvorby UDF preparsera
- viac flexibility

Ak hľadáte vhodnosť použitia fulltextového indexovania a vyhľadávania, tak asi najideálnejší príklad je v spojení s inter/intra-netom.

Naznačím príklad z praxe:

Denno-denne zbierame príspevky z rôznych emailových konferencií. Tie sa potom spracúvajú rôznymi programami a ukladajú do MySQL databáze. Užívateľ na klientskej stanici môže spustiť program, ktorý dokáže v tejto databáze vyhľadávať tie záznamy, ktoré obsahujú požadované slovo. Keďže sa tu nachádzajú informácie v textovej podobe a je potrebné vyhľadávať v celom texte, práve tu sa uplatní fulltextové vyhľadávanie.

Avšak nesmieme zabudnúť, že fulltext index je pomerne nová vlastnosť MySQL a preto nemusí hneď naplniť naše očakávanie.

Isto poznáte situáciu, keď máme určitú tabuľku s dátami, ku ktorej pristupuje naraz niekoľko užívateľov tak, že jeden chce iba čítať, ale ten druhý chce zapísať nové dáta do tabuľky. Ako to urobiť, aby nedošlo k nejednoznačnosti dát? Na to slúži systém zamykania.

Ale ten si vysvetlíme nabadúce.

Malé veľké databázy III / 3.časť

Leto je v plnom prúde, mnohí z nás chodia k vode. Aj ja si chodím zaplávať medzi pirane na naše miestne plážovisko, pomenované podľa jednej farby vody (to akože aby som nerobil reklamu). A môj bicyklík som si zamykal zámkom. Ale jedného dňa sa mi zámok stratil, a tak je veru veľmi zle, keď si nemám čím svoj bicykel zamknúť. Jednoducho bez zámkov to v dnešnom svete nejde. Ale akú to má súvislosť s databázami? Aj tu sa používajú zámky! Neveríte?

Vlákno

Doteraz sme tvorili aplikácie, ku ktorým pristupovali jednotliví klienti akosi sporadicky, teda nie súčasne. Ale v praxi sa stáva pravý opak.

Predstavme si situáciu, že sme vlastníkmí centrálného obchodu s elektronikou a vlastnime (okrem bavoráka, chichichi) aj účtovný systém na báze SQL, cez ktorý si jednotliví obchodníci objednávajú u nás tovar. Je samozrejmé, že každý obchodník sídli v inej časti krajiny a do nášho systému sa pripája -akože inak - z diaľky. Keďže sme veľmi prosperujúci veľkosklad, naša databáza je teda riadne vyťažená. A to tak, že sa veľmi často stáva, že klienti posielajú svoje požiadavky skoro súčasne. Zatiaľ čo sa jedna požiadavka snaží dáta z databázy čítať, iná sa pokúša zapisovať a ďalšie na túto možnosť len čakajú.

Každá požiadavka, prichádzajúca do MySQL servera od iného klienta, prichádza akoby po zvláštnom kanále, ktorý si môžeme predstaviť ako nejaký drôt alebo vlákno. V odbornej terminológii sa tomuto kanálu skutočne hovorí **vlákno** (angl. *thread*). Vlákno vznikne pri nadviazaní spojenia a po ukončení požiadavky automaticky zanikne. MySQL server môže obsluhovať naraz niekoľko vlákien, dokonca aj vo vzťahu k jednej tabuľke. Všetky vlákna sa správajú tak, aby (pokiaľ možno) svojou činnosťou nenarušili činnosť iných vlákien.

Poznámka:

Aj keď sa zdá, že podstata vlákien je podobná procesom v operačnom systéme, rozdiel tu je! Služba - démon MySQL je sám o sebe jedným z procesov, bežiacich v OS, ktorý môže obsahovať naraz mnoho vlákien.

Príklad č.I

Nech v našom účtovnom systéme máme tabuľku **Zasoby**, obsahujúcu informácie o stave zásob. Tabuľka obsahuje položku **sklad_pocet**, kde sú výšky zásob uložené.

Nech v danom okamžiku obchodník č.1 z východnej časti krajiny pomocou klientskej aplikácie objedná určité množstvo - pre tento prípad 1 ks - televízneho prijímača.

Serverová aplikácia zistí, či sa na sklade nachádza dostatočné množstvo vyžiadaného tovaru. Objednávka je prijatá za predpokladu, že stav zásob je vyšší alebo rovný požiadavke obchodníka č.1. Na záver sa zníži stav zásob o objednaný počet (teda v tomto prípade o jeden kus).

Naznačme si sekvenciu SQL príkazov, ktoré budú jadrom aplikácie (premenné sú len ilustratívne):

```
SELECT sklad_pocet FROM zasoby WHERE id_tovaru = id_objed_tovaru
```

kód, ktorý overí dostatok tovaru na sklade

ak áno, potom prijme objednávku a aktualizuje počet zásob...

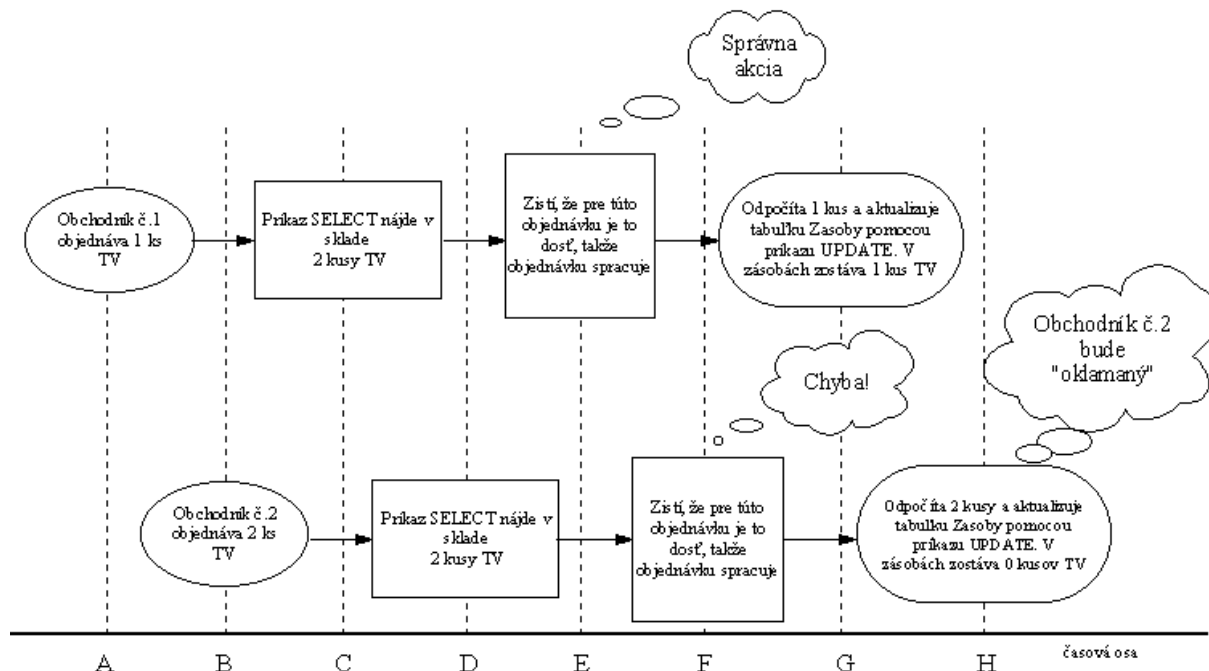
```
UPDATE Zasoby SET sklad_pocet = mnozstvo_zo_selectu - mnozstvo_obj_tovaru
WHERE id_tovaru = id_objed_tovaru
```

Medzitým, čo sa tieto riadky vykonávajú, objednáva rovnaký tovar úplne iný obchodník č.2 zo západnej časti krajiny. Rovnaký blok kódu sa začne vykonávať, ale už na inom vlákne.

Čo sa stane?

V tomto prípade si obe vlákna vzájomne narušia výpočty. Nesprávny predpoklad niektorého z vlákien môže mať za následok zápis chybného údaju do tabuľky.

Pozrime sa na obrázok č. III-1:



Ak bude náš systém fungovať takto, pravdepodobne obchodníka č.2 sklameme.

Prečo?

Obchodník č.1 si v okamžiku A objedná 1 ks televízora. Aplikácia sa v okamžiku C pozrie (príkazom *SELECT*) do skladových zásob a zistí, že v sklade sú ešte dva televízory. V okamžiku E na základe výšky zásob (keďže požiadavka je iba na jeden, tak je objednávka prijatá) začne objednávku spracovávať a aplikácia (príkazom *UPDATE*) upraví výšku zásob na zostávajúci jeden kus v okamžiku G.

Skoro súčasne, ale predsa o chvíľku neskôr obchodník č.2 v časovom úseku B objedná 2 ks televízorov.

Aplikácia (na inom vlákne!) v okamžiku D zo zásob zistí, že na sklade sú ešte 2 televízne prijímače (počet 2 tam je preto, lebo prvé vlákno ešte neodpísalo 1 ks TV objednaný obchodníkom č.1. To urobí až nasledujúci krok - v okamžiku E!). Takže toto vlákno stále počíta z dvomi kusmi TV! V okamžiku F vlákno zistí, že pre objednávku č.2 je to stále dosť, takže ju začne spracovávať a v okamžiku H odpočíta od výsledku príkazu *SELECT* 2 ks a príkazom *UPDATE* zapíše, že na sklade zostáva 0 (2-2) kusov TV. Aj keď bude objednávka obchodníka č.2 prijatá, v skutočnosti zostane nesplnená, lebo my fyzicky nemáme 3 televízne prijímače.

Logicky vieme, že to nie je pravda. Veď predsa na začiatku oboch operácií boli v sklade len 2 ks. Obchodník č. 1 si objednal 1 kus a obchodník č.2 2 kusy, takže nemôžu byť obidvaja uspokojení a zároveň v sklade zostať nula kusov.

Kde nastala chyba?

Práve medzi úsekmi D až G. V tej dobe druhé vlákno príkazom *SELECT* načítalo stav zásob, ktorý nebol ešte upravený - aktualizovaný prvým vláknom príkazom *UPDATE*.

Príklad č.II

Riešením tohoto problému sú **zámky**. Nie také bicyklové, ale programové. Zamykanie tabuliek poskytuje vláknám **výhradný prístup** k zdrojovým tabuľkám. Výhradný prístup znamená, že záznam, ktorý práve upravujeme, nemôže zmeniť žiadny iný užívateľ. Po dokončení úprav je zámok uvoľnený a prístup k záznamu môžu získať ostatné vlákna iných užívateľov.

Poznámka:

Typy zamykania v rôznych systémoch SQL môžu byť rôzne. Spravidla sa nemusí zamykať celá tabuľka, ale sa zamkne iba požadovaný riadok v tabuľke, ba dokonca iba jedno políčko v ňom. MySQL dokáže zamykať iba celú tabuľku naraz. Mnohí ne-maj-es-kvé-el-áci to tejto databáze vyčítajú ako nedostatok. Ale vzhľadom na jej nepomernú rýchlosť sa to v praxi neprejaví ako veľké zdržanie.

V MySQL sa tabuľka zamyká príkazom

LOCK TABLES meno_tabuľky (AS alias) {READ | [LOW_PRIORITY] | WRITE}

Príkaz **LOCK TABLES** je možné použiť na viac tabuliek naraz, kde sa jednotlivé zápisy oddeľujú čiarkou.

V MySQL existujú dva druhy zámok - zámok pre **čítanie** (*READ*) a zámok pre **zápis** (*WRITE*).

Pri práci s týmito zámkami platia tieto nasledujúce zásady:

- Ø Ak niektoré vlákno zamkne tabuľku pre čítanie (zámok typu *READ*), môže z nej toto vlákno, ako aj všetky ostatné vlákna iba čítať.
- Ø Ak niektoré vlákno zamkne tabuľku pre zápis (zámok typu *WRITE*), získa k nej výhradný prístup, teda len toto vlákno môže z danej tabuľky čítať alebo do nej zapisovať. Ostatné vlákna musia čakať (nemôžu ani čítať ani zapisovať), pokiaľ toto vlákno zámky neodstráni.

Odomknutie tabuľky sa vykoná príkazom **UNLOCK TABLES**.

Prepracujme výpis v prvého príkladu tak, aby sme využili zámky:

LOCK TABLES Zasoby WRITE

```
SELECT sklad_pocet FROM Zasoby WHERE id_tovaru = id_objed_tovaru
```

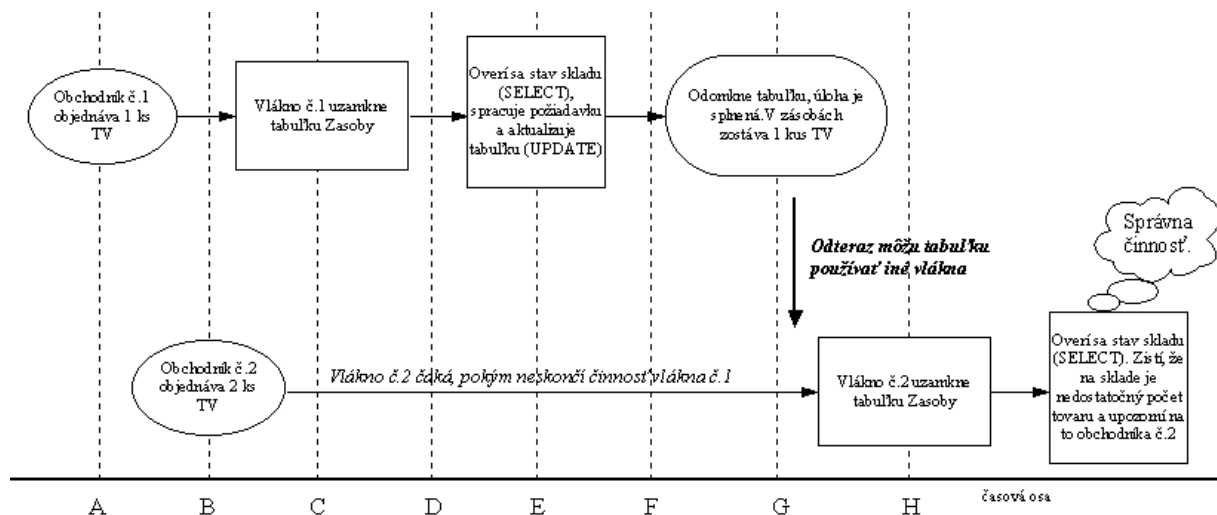
kód, ktorý overí dostatok tovaru na sklade
ak áno, potom prijme objednávku a aktualizuje počet zásob...

```
UPDATE Zasoby SET sklad_pocet = mnozstvo_zo_selectu - mnozstvo_obj_tovaru  
WHERE id_tovaru = id_objed_tovaru
```

UNLOCK TABLES

Teraz bude aktualizácia záznamov v tabuľke určitým spôsobom riadená, lebo vlákno obchodníka č.2 musí čakať, pokiaľ prvé vlákno nedokončí započaté zmeny.

Pozrime sa na obrázok č.III-2:



Vlákno obchodníka č. 1 je podobné, ako v predchádzajúcom príklade, len v okamžiku C pribudol zámok na tabuľku **Zasoby**.

Keď však obchodník č.2 v okamžiku B vyšle objednávku na 2 kusy televízneho prijímača, druhé vlákno, ktoré má spracovávať túto úlohu, musí čakať, pokiaľ neskončí akcia prvého vlákna, teda pokiaľ prvé vlákno nezaktualizuje stav zásob na 1 kus a neodomkne príslušnú tabuľku. To sa stane až v okamžiku G.

Až v tomto momente začne pracovať druhé vlákno a v okamžiku *H* pre zmenu ono zamkne tabuľku pre seba. V následnom okamžiku však zistí, že zákazník síce požaduje 2 kusy televízorov, ale na sklade je iba 1 kus, čo nestačí! Objednávku nevyrieši, ale upozorní obchodníka č.2, že jeho objednávka nemôže byť uspokojená z nedostatku tovaru.

Pekné, nie?

Príklad č.III

V predchádzajúcich príkladoch sme používali iba jednu tabuľku - **Zasoby**. Dopyty - *queries* - však bežne pracujú s viacerými tabuľkami. Postupnosť dopytov pri jednej operácii sa môže niekedy skladať aj z niekoľkých krokov. V takom prípade musíme pred započatím úprav zamknúť všetky príslušné tabuľky.

Predstavme si, že náš kód je už prepracovanejší a že chceme do tabuľky obchodných transakcií s názvom **Objednavky** pridať záznam o práve prebiehajúcej objednávke. Ako nový záznam tabuľky **Objednavky** budeme pridávať meno a adresu zákazníka.

Postupnosť SQL príkazov by mohla byť potom takáto:

```
LOCK TABLES Zasoby WRITE, Objednavky WRITE, Zakaznici READ
```

```
SELECT sklad_pocet FROM Zasoby WHERE id_tovaru = id_objed_tovaru
```

kód, ktorý overí dostatok tovaru na sklade

ak áno, potom prijme objednávku a aktualizuje počet zásob...

```
UPDATE Zasoby SET sklad_pocet = mnozstvo_zo_selectu - mnozstvo_obj_tovaru
WHERE id_tovaru = id_objed_tovaru
```

```
SELECT meno, adresa FROM Zakaznici WHERE id_zakaznika = id_objednavajuceho_zakaznika
```

vloženie získaných údajov do premenných *meno_z* a *adresa_z*

```
INSERT INTO Objednavky VALUES (meno_z, adresa_z)
```

```
UNLOCK TABLES
```

Všimnime si, že sme tabuľky **Zasoby** a **Objednavky** uzamkli pre zápis, zatiaľčo tabuľku **Zakaznici** sme zamkli iba pre čítanie.

Prečo?

Túto tabuľku (**Zakaznici**) nemusíme zamykať pre výhradný prístup, pretože do nej nebudeme nič zapisovať.

Ostatné vlákna budú môcť z nej čítať, aj keď sme ju uzamkli.

Zapamätajte si!

Ak z tabuľky iba čítame (príkaz SELECT), môžeme ju zamknúť iba pre zápis (READ).

Pre ostatné SQL príkazy (INSERT, UPDATE, DELETE) musíme použiť zámok pre zápis (WRITE).

Uviaznutie

Uvedomme si, že všetky tabuľky, s ktorými v našom projekte pracujeme, musia byť uzamknuté od počiatku úprav až do ich konca. Zabráni to uviaznutiu alebo zablokovaniu, čo je situácia, pri ktorej sú trvale zablokované dve či viac vlákien a každé vlákno čaká na tabuľku, ktorá je výlučne držaná iným zablokovaným vláknom. Keby sme neuzamkli všetky príslušné tabuľky, mohli by všetky aktívne vlákna začať operáciu, ale žiadne by ju nemohlo dokončiť. Predpokladajme situáciu, keď vlákno č.1 zamkne tabuľku **A** a vlákno č.2 zamkne tabuľku **B**. Vlákno č.2 však potrebuje k dokončeniu operácie údaje z tabuľky **A** (ktorú si uzamklo vlákno č.1). Musí teda čakať, pokiaľ ju vlákno č.1 neuvoľní. K tomu však nemusí dôjsť. Môže sa totiž stať, že práve vlákno č.1 tiež čaká na uvoľnenie tabuľky **B**. Tejto situácii sa hovorí **uviaznutie** a v odbornej angličtine sa uvádza ako **deadlock** (*smrteľný zámok*).

Fronty

Zadanie príkazu **LOCK TABLES** v požiadavke na uzamknutie ešte neznamená, že sa uzamknutie podarí! Pred uzamknutím tabuľky pre výhradný alebo zdieľaný prístup musí najprv prebehnúť na MySQL serveri zložitý

proces zaradenia do **fronty požiadaviek**. Na uzamknutie tabuľky musí vlákno spravidla určitú dobu počkať (jedná sa len o zlomky sekund!!!)

Mechанизmus radenia do front existuje preto, aby požiadavky na uzamknutie tabuľky mohli túto určitú dobu počkať, pokým iné vlákna požadované tabuľky neuvoľnia. Pre zámky typu *READ* a *WRITE* existujú dve samostatné fronty, ktoré existujú na odlišných princípoch.

Pre pokus o uzamknutie tabuľky s výhradným prístupom *WRITE* platí nasledujúci postup:

- Ø Ak nie je ešte požadovaná tabuľka zamknutá, je možné ju zamknúť zámkom typu *WRITE* pre výhradný prístup bez zaradenia do fronty
- Ø V opačnom prípade je požiadavka zaradená do fronty *WRITE*

Pre pokus o uzamknutie tabuľky pre čítanie *READ* platí nasledujúci postup:

- Ø Ak s tabuľkou nepracujú iné vlákna v režime výhradného prístupu *WRITE*, môžeme tabuľku zamknúť zámkom *READ* bez zaradenia do fronty
- Ø v opačnom prípade je požiadavka zaradená do fronty *READ*

Pri každom uvoľnení tabuľky dôjde k rozdeľovaniu prístupu. Pozor! Požiadavky vo fronte *WRITE* majú prednosť pred požiadavkami vo fronte *READ*! V prípade požiadavky uzamknutia tabuľky s výhradným prístupom sú teda omeškania minimálne.

MySQL umožní zamknúť tabuľku pre čítanie (*READ*) iba vtedy, ak už nečaká žiadna požiadavka vo fronte *WRITE*.

Je to z toho dôvodu, že pre väčšinu aplikácií je najdôležitejšia aktualizácia dát! Ak by sme z určitých dôvodov chceli uprednostniť čítanie pred zápisom, môžeme toto zmeniť pomocou predvoľby *LOW_PRIORITY WRITE*.

Ak vytvárame aplikáciu s veľkým vyťažením prúdov požiadaviek *READ*, nesmieme zabudnúť zaistiť dostatočný priestor pre vykonanie požiadavkov *WRITE*. Inak by sa mohlo stať, že pre samé čítanie nebude času na aktualizáciu databáz a užívatelia budú dostávať neaktuálne informácie.

Niekedy bude veľmi potrebné čítať z tabuľky, ktorú si iné vlákno výhradne zamklo pre svoj zápis zámkom *WRITE*. Na to slúži parameter *SELECT HIGH_PRIORITY*. Tento príkaz však používajme iba vo veľmi vážnych prípadoch.

Odomknutie tabuliek

Príkazom **UNLOCK TABLES** odomkneme všetky tabuľky, ktoré sme príslušným vláknom zamkli. Tabuľky sa automaticky odomknú aj v týchto prípadoch:

- Ø ak dané vlákno pošle ďalšiu požiadavku **LOCK TABLES**
- Ø ak sa spojenie s databázovým serverom preruší

Uzamknutie tabuliek nie je omedzené žiadnym časovým limitom!

Ako používať príkaz LOCK TABLES

V MySQL existuje niekoľko dobrých dôvodov, prečo by sme mali tabuľky pred použitím zamykať:

Ø *Viacnásobný prístup k tabuľke*

Vlákno môže tabuľky zamknúť a tým zaistiť, že bude mať k vybraným tabuľkám výhradný prístup. Práca vo výhradnom režime mu zaručuje, že môže započaté operácie dokončiť bez rizika, že sa upravované záznamy pokúsí medzitým pozmeniť niektoré iné vlákno. Čím viac príkazov SQL príslušný kód obsahuje, tým viac je zamykanie tabuliek dôležitejšie. Možnosť vzniku konfliktov je najzávažnejší dôvod zamykania.

Ø *Výkon*

Aplikácia môže v určitých prípadoch vyžadovať vykonanie niekoľkoriadkových operácií vo viacerých tabuľkách. Aplikácia bude rýchlejšia, ak sa pred spustením procedúry vybrané tabuľky uzamknú. Zaistiť sa tak úspešné dokončenie započatej operácie.

Najväčší dopad na výkon majú viacnásobné výskyt príkazu *INSERT*. Obyčajne sa vyrovnávacia pamäť indexu vyprázdni po vykonaní každého pridávacieho príkazu. Ak vybrané tabuľky zamkneme, vyrovnávacia pamäť sa vyprázdni až po vykonaní všetkých príkazov *INSERT* a následnom odomknutí tabuliek.

Ø *Chýba transakčné spracovanie*

MySQL neobsahovala do verzie 4 transakčné spracovanie. Toto je možné vďaka zámkom čiastočne nahradiť nasledujúcim postupom: Po zamknutí vybraných tabuliek overíme všetky podmienky, ktoré by mohli ovplyvniť úspech operácie. Ak je všetko v poriadku, uložíme zmeny. Na záver odomkneme tabuľky.

Ø *Zálohovanie databáze*

Pri práci s databázami by sme mali svoje dáta zálohovať a mať tak v dosahu konzistentnú a ucelenú kópiu aktuálneho stavu. Konzistencia je veľmi dôležitá. To znamená, že pri vytváraní záloh by nemalo byť žiadne vlákno aktívne. Pred vytváraním zálohy, ak už nie je vôbec možné „odpojiť“ databázu, by sme mali aspoň zamknúť zálohované tabuľky pre čítanie.

No, o zálohovaní sa dá toho povedať viac, a preto sa budeme touto témou zaoberať nabadúce.

Malé veľké databázy III / 4.časť

Už ste niekedy prišli o svoje drahocenné dáta? Že nie? Tak to ste šťastlivci! Ale každého to raz čaká, záleží len, ako bude na túto veľmi nepríjemnú skutočnosť pripravený. Kvalitná príprava umožní čo najmenej bolestný návrat k normálnej činnosti SQL serveru. Dnes si povieme, ako sa vhodne pripraviť na nechcenú stratu dát.

Ochrana a údržba dát (OUD) nie je veľmi populárna oblasť správy nielen SQL servera a databáz, ale aj samotných operačných systémov. To preto, lebo vyžaduje určité finančné aj ľudské zdroje bez zjavného vonkajšieho efektu. A preto sa táto oblasť aj veľmi zanedbáva v heslom „Nám sa to nemôže stať!“.

Do prvého veľkého problému!

Ochranu a údržbu dát si môžeme pripodobniť ochrane a údržbe majetku. Je to dobré prirovnanie, lebo dáta sú naozaj našim majetkom a ak si spomeniete na úplne prvú vetu tohoto celého seriálu - „Najväčšia cena je cena informácie“, budete so mnou súhlasiť. Veď aj náš fyzický majetok si chránime aj za cenu zvýšených nákladov na zámky či iné bezpečnostné zariadenia.

Ak sa vám stále zdá, že táto téma je úplne zbytočná, vráťte sa k nej, až o tie dáta naozaj prídete. Až stratíte informácie, ktoré ste pracne zbierali celé roky, keď sa vám zosype účtovníctvo alebo - nedajbože!- niekto vaše dáta zneužije v obchodnom styku či kriminalizujúcom prostredí. Len aby už nebolo neskoro...

Prevenencia a obnova

Prevenencia je činnosť, ktorá má zabrániť prípadnej strate dát a obnova je taká činnosť po strate dát, ktorá vedie k uvedeniu do pôvodného stavu pred stratou.

Ako v každej ľudskej činnosti, prevencia, akokoľvek nákladná (vo finančnom aj personálnom smere) je VŽDY lacnejšia ako obnova už spôsobených škôd. Preto ťažisko činnosti, spojených s OUD, leží v prevencii.

Medzi preventívne opatrenia patrí:

- Ø fyzická bezpečnosť
- Ø personálna bezpečnosť
- Ø ochrana dát

Pod pojmom obnovy dát si môžeme predstaviť

- Ø obnova samotných dát po strate
- Ø oprava poškodených dát

Fyzická bezpečnosť

Tejto oblasti sa venuje najmenšia pozornosť. Je skutočne veľmi zanedbávaná, ale ako uvidíme ďalej, je veľmi dôležitá.

Prvý krok pri fyzickom zabezpečení našich systémov spočíva vo vytvorení akéhosi plánu, v ktorom zadefinujeme naše nároky na fyzickú bezpečnosť. Vo firemnom prostredí alebo v prostredí štátnej a verejnej správy sa takýto plán tvorí vo forme smernice a spravidla sa tým niektoré firmy aj solídne živia. Na Slovensku sú k tomu vydané veľmi prísne zákony a príslušné smernice. Uvádzam to tu preto, lebo ak by ste chceli svoj dobrý produkt ponúknuť týmto zložkám, musíte počítať s týmto plánom. Pre naše domáce potreby či potreby malej súkromnej firmy bude postačovať, ak si tu uvedené jednotlivé body premyslíme a prípadne odstránime nedostatky.

Na začiatok si položíme nasledujúce otázky:

- Ø má niekto iný než my fyzický prístup k nášmu počítaču s SQL serverom?
- Ø čo ak to bude blázon alebo šialenec a pustí sa do neho kladivom?
- Ø čo by sa stalo, ak by k nám úplne náhodou neohlásene prišiel niekto od konkurencie?
- Ø ak budovu, kde je server postihne katastrofa, budeme môcť ďalej používať systém?
- Ø ak dôjde k havárii systému, ako sa budeme baviť so všetkými rozzúrenými užívateľmi a šéfmi?

Fyzická bezpečnosť má tieto prvky:

- Ø **Prostredie** - počítače sú pomerne citlivé zariadenia na prostredie, v ktorom pracujú. Prehliadnutie niektorého faktora môže viesť k často nepredvídateľnému zlyhaniu počítača. Faktory, ktoré negatívne ovplyvňujú činnosť počítača, sú:

- *ohneň* - dokáže zničiť počítače dokonale. Ak by aj samotný počítač oheň prežil, nakoniec ho zničí voda pri hasení. Premyslime si, ako sme na prípadný požiar pripravení, či je poruke vhodný hasiaci prístroj, či existuje iné požiarne zabezpečenie v okolí
- *dym* - je to silné abrazívum a zhromažďuje sa na hlavičkách diskov alebo iných mechanických prvkoch
- *prach* - tu platí to samé ako o dyme, plus to, že prach býva často elektricky vodivý
- *zemetrasenie* - v našich podmienkach málo aktuálne, ale treba s ním počítať
- *výbuch* - výbuch plynu alebo skladu horľavých látok spôsobí totálne zničenie servera
- *teplotné výkyvy* - počítač je náchylný na teplotné výkyvy. Tepelný šok spôsobí poškodenie elektroniky počítača.
- *hmyz a hlodavce* - ešte vám nikdy myš neprehryzla elektrické vodiče?
- *elektrické rušenie* - často známy jav hlavne vo výrobných priestoroch s elektrickými motormi. Pomôcť by mohla UPS-ka alebo napäťové filtre
- *blesky a výboje* - dokážu zničiť dáta na magnetických nosičoch. Je vhodné uchovávať tieto média v kovových ochranných skrinkách
- *vibrácie* - od prípadných vibrácií si pomôžeme gumovou podložkou
- *voda a vlhkosť* - je najväčší nepriateľ elektroniky. Spôsobuje koróziu a drobné skraty s veľkými dôsledkami
- *jedlo a pitie* - nesmejte sa! Vyliata káva alebo kus šunky v klávesnici vás asi nepoteší. Zvlášť, ak sa závada neprejaví hneď, ale až keď kávačka zatečie na to správne miesto!

- Ø **Fyzický prístup** - zdravý rozum hovorí, že počítače by mali byť v uzamknutej miestnosti. Je ale uzamknutá miestnosť bezpečná? Čo také
 - *zvýšené podlahy a znížené stropy* - nemôže niekto vojsť tadiaľto? Stačí len potkan alebo myš!
 - *prístup cez vetracie šachty* - mali by byť opatrené sieťkami proti spomínaným hlodavcom
- Ø **Vandalizmus** - je nezanedbateľný prvok. Úmyselné poškodenie počítača alebo kabeláže dokáže znefunkčniť náš systém na niekoľko hodín či dní. Nikdy nepodceňujte človeka, ktorý má ľubovoľný, aj patologický dôvod sa vám pomstiť! Uložte kable z dosahu alebo aspoň tak, aby neboli na očiach.
- Ø **Krádež** - asi najčastejšia nepríjemnosť. Tá môže byť vykonaná s úmyslom získať techniku ako cennosť alebo priamo za účelom získania dát. Vhodne skombinované opatrenia, podobné ochrane iného cenného majetku, zámkami počnúc a elektronickými systémami končiac zabráni nielen strate dát - tie obnovíme z prípadnej zálohy, ale zabráni zneužitiu dát.

Personálna bezpečnosť

Aj keď sa to nezdá, personálna bezpečnosť je rovnako dôležitá, ako fyzická. To preto, že človek dokáže ublížiť systému rovnako, ak nie viac, ako prírodné živly. Dobre poznáte svojich užívateľov? Dôverujete tomu, kto vás má pri údržbe systému zastupovať v prípade dovolenky alebo nemocnosti?

Pri personálnom zabezpečení je potrebné venovať pozornosť týmto aspektom:

- Ø **minulosť** - znie to paranoidne, ale preverte, či dotyčná osoba, ktorá k vám nastúpila do zamestnania a tvrdo sa dožaduje prístupu k vašim firemným dátam :
 - *má príslušné vzdelanie a schopnosti*. Taký „chrobák truhlík“, čo všade bol a všetko vie, dokáže urobiť značnú paseku v systéme.
 - *nepochádza z konkurenčného prostredia*. Čo ak je nasadený na získanie vašich dát?
- Ø **vstupné školenia** - mali by obsahovať základné návyky bezpečnostných opatrení - od používania hesiel, cez ukončovanie sedenia, zálohovanie dát až po zamykanie kancelárie. Pre tých, čo pracujú v právnom prostredí, nechajte si podpísať, že používatelia boli riadne poučení. (mám nedobru skúsenosť!!!)
- Ø **priebežné školenia** - zabezpečia pravidelné oboznamovanie vašich užívateľov s novinkami vo vašom systéme. A nezabudnite - „*Repetitio est mater studiorum*“ (Opakovanie je matka múdrosti).
- Ø **sledovanie prístupu** - je dobré, keď budú používatelia vedieť, že sledujete ich prístupy do systému. Aspoň nebudú pokúšať!
- Ø **minimalizácia privilégii** - dajte jednotlivým používateľom len najnutnejšie prístupové práva. Nespôsobia škody, ktoré nechcete. A to ani nechcenou činnosťou alebo nesprávnymi postupmi.
- Ø **odchod používateľa** - vždy po odchode používateľa z organizácie odstráňte jeho účet zo systému. Zabráňte zneužitiu. Zvláštnu pozornosť venujte používateľom, ktorí zastávali kľúčové pozície, mali značné prístupové práva a odchádzajú dobrovoľne! Verte mi, tento jav je dnes bežný a niet lepšej časovanej bomby!

Ochrana dát

Ochrana dát je azda najdôležitejším prvkom prevencie pred stratami inofmácií. Ak aj dokážeme zabezpečiť všetky body z fyzickej a personálnej bezpečnosti, môžeme očakávať, že dôjde k poškodeniu dát inými činiteľmi, ako je „seknutie“ systému, nechcený výmaz (aj zo strany roota!), nezabezpečením integrity (zlým návrhom celej aplikácie - niečo sa niekde vymazalo, čo sa nemalo) alebo úmyselné poškodenie samotných dát . Ochrana dát spočíva v týchto bodoch:

- Ø zálohovanie systému
- Ø zálohovanie a obnova dát
- Ø ochrana záloh dát

Zálohovanie systému

Po týmto pojmom si môžeme predstaviť fyzickú - hmotnú zálohu celého systému, teda vytvorenie záložného servera na fyzicky inom počítači. Táto varianta je finančne náročná a dovoľia si ju len solventnejšie firmy či organizácie. U nás prevádzkujeme trojnásobnú zálohu systému - 1 hlavný linuxový server, 1 záložný linuxový server a 1 počítač so systémom Windows, kde je tiež nainštalovaná zodpovedajúca verzia MySQL. V prípade zlyhania linuxov (hm, hm, že by padli obidva naraz?) sme schopní behom niekoľkých minút aktivovať windowsovskú verziu MySQL a zo záloh dát sme schopní pripraviť náš projekt k činnosti. Ak nemáme dostatok prostriedkov na výstavbu druhého záložného linuxu, môžeme použiť variantu 1 hlavný server a 1 počítač s operačným systémom Windows, napr. 95 či 98. Môže to byť aj ľubovoľná pracovná stanica, ktorá by dočasne prevzala úlohu servera. Vzhľadom k tomu, že sme sa naučili nainštalovať a spustiť “svätú trojicu” - MySQL - Apache - PHP pod obidvoma operačnými systémami, je to možné považovať za vyhovujúce riešenie. Tu je dôležitá ešte aj stratégia zálohovania. Aktualizácia dát na záložnom serveri môže byť:

- Ø synchronna
- Ø asynchronna

Synchronna aktualizácia dát je proces, ktorý zapisuje dáta do konkrétnej databáze na hlavnom serveri a zároveň do stanovenej databáze na záložnom serveri. MySQL takéto zálohovanie nepodporuje, aspoň nie priamo, aj keď ma napadajú spôsoby, ako to vyriešiť.

Asynchronna aktualizácia dát je proces, kde sa dáta zapisujú do určitej databáze na hlavnom systéme a na záložný systém sa prenesú až z určitým oneskorením. Tento systém je už použiteľný aj v MySQL, napr. pomocou démona **cron** v linuxovom prostredí. Ten v určitých časových intervaloch okopíruje dáta z hlavného servera do záložného. Ten sa takto stáva pripravený na prevzatie úlohy v prípade krachu hlavného servera.

Zálohovanie a obnova dát

je najvhodnejší spôsob, ako sa vrátiť k správnym dátam po ich poškodení. Rovnako dôležité je vytvorenie záložnej kópie, ale aj proces obnovy v prípade havárie. Tento proces musí byť rýchly a zároveň bezpečný, aby sme sa mohli čo najskôr vrátiť k normálnej prevádzke.

Proces vytvárania záloh má mnoho metodologických štúdií, ktoré vznikli ešte v dobách, keď žiadne magnetické médium nebolo dostatočne spoľahlivé. V dobách CD a DVD médií je podstata procesu zálohovania značne jednoduchšia. Cez spomínané zjednodušenie sú základné princípy platné aj dnes a celý proces záloh môžeme rozdeliť do týchto bodov:

- Ø úplné zálohy
- Ø inkrementálne zálohy

Keďže obidva spôsoby sú podporované v MySQL, pozrieme sa teraz na ne podrobnejšie.

Úplné zálohovanie

Pri tomto spôsobe zálohovania sa zálohujú všetky dáta vo všetkých databázach (prípadne vrátane prístupových práv, teda aj databáze **mysql**), ktoré systém MySQL obsahuje. Zálohovať môžeme na pevný disk, pásku, cédečko či iné médium, vrátane sieťových. Takáto záloha sa tvorí raz za určité obdobie, najčastejšie raz za týždeň.

Obnova dát je v tomto prípade veľmi jednoduchá - obnoví sa naraz celý databázový server vrátane prístupových práv. Nevýhodou je množstvo dát a s tým súvisiace časové zdržanie pri tvorbe zálohy. Doporučujem úplnú zálohu vykonávať na začiatku alebo konci pracovného týždňa a ako úložné médium použiť fyzicky iný disk, ako je ten pracovný. Je to pochopiteľné, lebo ak „kľakne“ disk s ostrými dátami, ku ktorým sa nedostaneme, nedostaneme sa ani k perfektne urobeným zálohám. Dnes, vzhľadom k dostupnosti a cenovej výhodnosti je dobré zvoliť CDR alebo CDRW médiá.

Inkrementačné zálohovanie

inak nazývané aj prírastkové - je také, kde sa zálohujú iba tie dáta, ktoré sa zmenili od poslednej úplnej zálohy. Takéto zálohovanie sa spravidla vykonáva denne. Jednotlivé prírastky sa ukladajú tak, aby inkrementačná záloha každého dňa bola na inom mieste média, napr. v inom, vhodne pomenovanom adresári alebo diskete. Ak nebudeme používať diskety, môžeme zvoliť aj pevný disk, či už sieťový (na inom počítači) alebo prinajhoršom vlastný. Predpokladajme, že keď dôjde k poruchám dát, nemusí to byť hneď porucha celého disku, takže sa k denným zálohám dostaneme. A keby nie, prinajhoršom sa vrátíme k stavu z predchádzajúceho týždňa, kedy sme robili úplnú zálohu na cédečko a to máme poruke v trezore.

Obnova dát je v tomto prípade zložitejšia. Najprv obnovíme dáta z poslednej úplnej zálohy a pokračujeme obnovou dát za jednotlivé dni, až dôjdeme k súčasnemu stavu.

Frekvencia tvorby záloh závisí od frekvencie zmien dát v danej databáze. Ak používame databázu len na čítanie, a jej dáta sa nemenia počas celého roka, nebudeme predsa robiť denné inkrementačné zálohy, neboli by vlastne žiadne. Ak sa nám menia dáta 1000 krát za deň, je vhodné robiť inkrementácie každú hodinu. Majme na pamäti, že obnovou dát vždy nejaké stratíme. Sú to tie, čo sa zmenili od poslednej zálohy. (Zákon schválnosti hovorí, že dáta sa porušia tesne pred vytvorením zálohy, nie hneď po nej, kedy ešte nedošlo k žiadnej zmene.) Musíme pamätať na to, že potrebujeme stratené dáta nahradiť a to najčastejšie ručne - opisom z papierovej predlohy a podobne. (Ak nám takéto riešenie nevyhovuje, musíme pristúpiť na podstatne nákladnejšiu synchronnú aktualizáciu.)

Azda napoužívanejšia schéma vytvárania záloh je takáto:

- Ø úplnú zálohu vytvárame mimo pracovnej doby raz týždenne, najlepšie v piatok popoludní alebo vo večerných hodinách. Preferujeme piatok, lebo keby niekde niečo v zálohovaní „kikslo“, máme celú sobotu a nedeľu na nápravu
- Ø inkrementačné (prírastkové, čiastkové) zálohy vykonávame denne, konkrétne raz za deň, mimo hlavného náporu požiadaviek prístupu k dátam, spravidla vo večerných hodinách a to tak, že zálohu z každého dňa uložíme separátne, aby sme neprepísali včerajšiu zálohu dnešnou.

Takto dosiahneme zálohu dát, ktorá je maximálne jeden deň stará.

Vačšina tejto činnosti sa dá zautomatizovať, obzvlášť v linuxe, a o výsledku sa môžeme nechať informovať mailom alebo esemeskou.

Spomeňme si na predchádzajú časť seriálu. Hovorili sme, že je dôležité, aby počas vytvárania záloh nedošlo k čiastočným zmenám dát. Preto by sme mali na dobu tvorby záloh databázu odpojiť, a ak to naozaj nie je možné, tak aspoň zálohované tabuľky zamknúť na čítanie!

Ochrana záloh dát

Nič nám nebude platný dokonalý a prepracovaný systém záloh dát, ak nezabezpečíme ochranu týchto záloh. Zbytočné budú zálohy na harddisku počítača, ak sa tento zrúti zároveň s ostrými dátami. Preto odkladajme zálohy na fyzicky iné médium. Taktiež nezabudnime, že aj diskety, pásky či cédečka môžu zhorieť zároveň s počítačom, ak boli uložené v tej istej miestnosti, kde samotný server. (No vysvetľujte šéfovi, že ste zálohy robili správne, len sú zhorené!). Nezabudnime na to, že zálohy môžu byť objektom záujmu iných osôb tak, ako ostré dáta. Ak sme perfektne zabezpečili serverovňu strážnymi psami a širokoplecými mužmi v čiernom, nevozme záložné kópie v aute za zadným sklom aj s popisom, že je to záloha databázy klientov nemenovanej banky za prvý kvartál tohto roku!

Pamätajte si! Na zálohy dát sa vzťahujú tie isté princípy ochrany, ako na samotné dáta!

A ešte jeden poznatok (zo života!):

Občas skontrolujme, či vytvorené zálohy sú naozaj použiteľné a čitateľné a či zvládame obnovu dát!

Nespoliehajte sa na to, že to overíme, až keď to budeme potrebovať. To aby sme sa nedočkali nemilého prekvapenia!

Ktoré súbory sú dôležité? To zistíme tak, že si položíme otázku: „*Strata ktorých dát ma bude bolieť?*“. A dajme si odpoveď (že asi všetky). A takto zodpovedané dáta **ZÁLOHUJME!**

Vraťme sa konečne k MySQL a povedzme si, ako budeme vykonávať jednotlivé zálohy dát a ich obnovu v tomto systéme.

Vytvorenie úplnej zálohy dát v MySQL

Na vytvorenie úplnej zálohy sa v MySQL používa utilita *mysqldump*. Jej úlohou je vygenerovať obsah databázy alebo tabuľky v tvare SQL príkazov do čisto textového súboru. Podľa nastavených prepínačov dokáže vytvoriť

príkazy pre návrh tabuľky a pre vkladanie príslušných dát. Obsah tohoto súboru je potom jednoducho použiteľný pri obnove dát.

Formálny zápis príkazu mysqldump je:

mysqldump [prepínače] -u *používateľ* -p*heslo* *meno_databázy* [*meno_tabuľky*] > *meno_súboru*

Parametre v hranatých zátvorkách sú nepovinné. Znak > (zobáčik) značí, že výstup programu *mysqldump* nebude vystupovať na obrazovku (štandardný výstup), ale sa presmeruje do súboru.

Predstavme si, že chceme z našej známej databáze **kniznica** vydumpovať tabuľku **KNIHA** do súboru **kniha.sql**. V takom prípade zapíšeme príkaz:

mysqldump -u root -p*heslo* *kniznica* *kniha* >kniha.sql

Takto vzniknutý súbor **kniha.sql** sa nachádza v ceste spúšťania podľa príslušných konvencií operačného systému. Ak ho chceme ukladať presne tam, kde potrebujeme, napíšeme úplnú cestu súboru.

Čo sa stalo?

Program *mysqldump* zistil štruktúru tabuľky **kniha** a jej obsah. Previedol ich do príkazov SQL a uložil do súboru **kniha.sql**. Jeho obsah je na výpise č.III-4-1:

```
# MySQL dump 8.19
#
# Host: localhost      Database: knihnica
#-----
# Server version  4.0.1-alpha
#
# Table structure for table 'kniha'
#

CREATE TABLE kniha (
  id int(11) NOT NULL auto_increment,
  nazov varchar(40) default NULL,
  autor varchar(30) default NULL,
  vydavatel varchar(25) default NULL,
  cis_odd int(11) default NULL,
  cena decimal(5,2) default NULL,
  poznamka varchar(25) default NULL,
  PRIMARY KEY (id)
) TYPE=MyISAM PACK_KEYS=1;

#
# Dumping data for table 'kniha'
#

INSERT INTO kniha VALUES (1,'Angelika a kral','Golon, Anne a
Serge','Slovensky spisovatel',2,56.00,'');
INSERT INTO kniha VALUES (2,'KGB','Gordijevsky,
Oleg','EAAP',6,239.00,'');
INSERT INTO kniha VALUES (3,'Bratia Ricovci','Simenon,
Georges','Smena',3,18.00,'');
INSERT INTO kniha VALUES (4,'Vtaky v trni','McCulloughova,
Collen','Slovensky spisovatel',2,66.00,'');
INSERT INTO kniha VALUES (5,'Linux - prakticky pruvodce','Sobell, Mark
G.','Computer Press',7,1073.00,'');
INSERT INTO kniha VALUES (6,'Naucte se programovat v Delphi','Binzinger,
Thomas','Grada',7,439.00,'');
```



```

INSERT INTO kniha VALUES (7,'Pouzivame linux','Welsh, M., Kaufman,
L.','Computer Press',7,494.00,'');
INSERT INTO kniha VALUES (8,'Z polovnickej kapsy','Moric, Rudo','Mlade
leta',4,89.00,'');
INSERT INTO kniha VALUES (9,'Plebejska kosela','Mihalik,
Vojtech','Slovensky spisovatel',1,15.00,'');
INSERT INTO kniha VALUES (10,'Europou bez penazi','Hlubucek, Petr,
Ing.','Roman Kasan',5,34.00,'');

```

Všimnime si, že sa tento súbor skladá z obidvoch spomínaných častí - vytvorenie tabuľky kniha príkazom *CREATE TABLE* a jej naplnenie príslušnými dátami príkazom *INSERT*.

Samotný príkaz *mysqldump* má neskonalé možnosti. Môžeme vytvoriť iba štruktúru tabuľky alebo len samotné dáta. Môžeme vytvoriť taký výpis, aby boli informácie prenosné do iného SQL systému. Možnosti sa ovládajú pomocou prepínačov, ktorých stručný prehľad je v tabuľke č.III-4-2:

Tabuľka č.III-4-2:

Prepínač	Skrátený zápis	Popis
--add-drop-table		Doplní obsah súboru o príkaz DROP TABLE IF EXIST, ktorý umiestni pred príkaz CREATE TABLE. To zabezpečí, že ak už tabuľka s rovnakým názvom existuje, odstráni ju a nahradí novou
--add - locks		Zaistí vloženie príkazov INSERT medzi príkazy LOCK TABLES a UNLOCK TABLES. Takto nedovolí používateľom manipulovať s tabuľkou, pokiaľ sa do nej nevložia všetky dáta pri obnove dát
--complete_insert	-c	Pridá názov ku všetkým poľam príkazu INSERT. Tento prepínač je vhodný najmä pri exporte dát do iného SQL systému
--flush - logs	-F	Pred vytvorením výpisu vyprázdni súbor protokolu služby
--force	-f	Zaistí, že mysqldump bude pokračovať vo výpise, aj keď medzitým dôjde k neočakávanej chybe
--lock-tables	-l	Uzamkne pred vytvorením výpisu všetky tabuľky, z ktorých sa výpis bude zostavovať
--no-create-info	-t	Zabráni tvorbe príkazu CREATE TABLE. Vhodné pokiaľ chceme získať iba výpis samotných dát
--no-data	-d	Zabráni tvorbe príkazu INSERT. Výpis bude obsahovať iba štruktúru tabuľky
--quick	-q	Nedovolí systému MySQL načítanie celého výpisu do operačnej pamäte pred fyzickým vytvorením súboru. Vynúti priamy zápis do súboru už počas čítania informácií z databázy
--opt		Zapne všetky predvoľby, ktoré urýchlia proces vytvárania výpisu
--tab=cesta	-T	Zaistí vytvorenie dvoch súborov. Jeden - s príponou .sql bude obsahovať SQL príkazy na vytvorenie štruktúry tabuľky CREATE TABLE a druhý - s príponou .txt iba výpis dát, oddelených tabulátorom (bez príkazu INSERT). Argument cesta určuje adrsár, kde budú príslušné súbory vytvorené. Uvedený adresár musí existovať pred spustením
--full		Doplní príkaz CREATE TABLE o ďalšie podrobnejšie informácie
--delayed-insert		Vloží do príkazu INSERT kľúčové slovo DELAYED
--where = "podmienka WHERE"	-w "podmienka where"	Umožňuje filtrovanie záznamov, ktoré sa vyexportujú do výpisu na základe splnenia príslušnej podmienky

Vyskúšajme tento príkaz:

mysqldump --tab=d:\ -u root -p heslo kniznica zaner

V koreni disku D: (linuxáci v príslušnej časti stromu) sa vytvoria súbory **zaner.sql** (výpis č. III-4-3):

```
# MySQL dump 8.19
#
# Host: localhost      Database: kniznica
#-----
# Server version  4.0.1-alpha
#
# Table structure for table 'zaner'
#

CREATE TABLE zaner (
  cis_odd int(11) NOT NULL auto_increment,
  tematika varchar(20) default NULL,
  PRIMARY KEY  (cis_odd)
) TYPE=MyISAM PACK_KEYS=1;
```

súbor **zaner.txt** (výpis č.III-4-4 pozri úlohu prepínača **tab**):

```
1      poezia
2      roman
3      krimi
4      detska lit.
5      cestopis
6      lit. faktu
7      odborná lit.
```

Použitie *mysqldump* v základnom režime, teda bez prepínačov, plne vyhoví bežnej zálohovacej stratégii. Takto vytvorenú úplnú zálohu tabuľky, databáze alebo celého MySQL stroja odložíme tak, ako sme si to povedali skôr.

Ak si tieto príkazy odskúšame, zistíme, že sa vykonajú pomerne rýchlo a ani nezaberajú tak veľa miesta. Prečo teda potom robiť inkrementálne zálohy a nepoužívať iba úplnú zálohu?

To záleží na veľkosti dát. Pokiaľ máme tabuľku s tisícmi záznamami, postačuje vykonávanie kompletnej zálohy bez použitia inkrementálnych záloh.

Ak však máme databázu s miliónmi záznamov a zmeny sa robia tisíc krát za deň, bez klasickej zálohovej stratégie sa asi nezaobídeme. Ak sa úplná záloha dumpuje niekoľko hodín, nebudeme ju chcieť robiť každý deň. Ak k tomu pripočítame fakt, že je nutné na dobu tvorby záloh databázu odpojiť alebo aspoň obmedziť do nej zápis, musíme vykonávať inkrementálne zálohovanie, ktoré je časovo menej náročné.

Vytvorenie inkrementálnej zálohy dát v MySQL

Tvorba inkrementálnych záloh v MySQL spočíva vo využití protokolovania činnosti na MySQL serveri.

Tie sa zapisujú do príslušných logovacích súborov vo forme SQL príkazov, ktoré menia hodnotu dát (*INSERT*, *UPDATE* a *DELETE*).

Aby sme mohli túto vlastnosť servera využiť, musíme ho spustiť s parametrom **--log-update** takto:

- pre linux : **safe_mysqld --log-update &**

- pre Windows: **mysqld --log-update**

Démon alebo služba vytvorí v príslušnom adresári logovací súbor, do ktorého bude protokolovať všetky činnosti, týkajúce sa manipulácie a zmeny dát.

Pomenovanie logovacieho súboru sa riadi ustálenou konvenciou **meno_servera.n** (napr. *mior.001*), kde **n** je číslo, ktoré narastá pri vytvorení každého nového protokolu. Nový protokolovací súbor sa vytvorí pri spustení alebo reštartovaní servera alebo pri spustení príkazu **mysqladmin** s parametrom **flush-logs** alebo **reload**.

Tak ako všetko v slušných programoch, tak aj názov protokolovacieho súboru sa dá zmeniť, napr.

mysqld --log-update=zaloha

Vytvorí sa súbor **zaloha.001**. Ak reštartujeme server, vytvorí sa nový súbor **zaloha.002**, do ktorého sa teraz budú ukladať príslušné príkazy. Tým zabezpečíme, aby sme získali zálohu z iného dňa do iného súboru.

Ak spustíme server tak, že protokolovaciemu súboru stanovíme príponu, napríklad **.log** takto:

mysqld --log-update=zaloha.log

vytvorí sa iba jeden súbor s menom **zaloha.log** a nikdy, ani po reštarte MySQL sa nevytvorí iný, ale zmeny dát sa pripisujú do tohoto súboru.

Aby sme zapísali všetky protokolované príkazy, musíme použiť príkaz *mysqladmin* s parametrom *flush-logs*. Tým sa vyprázdni pamäť, kde sa tieto protokolované logy dočasne ukladali. Zároveň sa vytvorí nový protokolovací súbor s číslom o jednotku väčším.

Našou úlohou je, aby sme zabezpečili správnu rotáciu súborov. V operačnom systéme Windows mnoho možností nemáme.

Naopak, v linuxe a unixe všeobecne, je možné využiť mnoho variant. Buď využijeme možnosti samotného operačného systému alebo môžeme využiť skript **mysql-log-rotate**, ktorý je pripravený v linuxovej verzii MySQL v adresári *support-files*.

Teraz vieme, že na sedem dní inkrementálnych záloh postačuje sedem súborov. Aby sme zbytočne nezaplňali disk inkrementálnymi súbormi donekonečna, dobre zvolené rotovanie alebo napísaný skriptík zabezpečí, že sa súbory, staršie ako sedem dní automaticky zmažú.

Tak ako úplné zálohovanie, tak aj inkrementálne vrátane rotácie logov môžeme - hlavne v linuxe - zautomatizovať a o výsledkoch alebo prípadných chybách sa necháme informovať.

Nesmieme zabudnúť, že technika je síce mocná čarodejnica, ale ľudský zásah je niekedy nevyhnutný, a to pri vložení výmenného záznamového média do záznamovej jednotky (disketa, páska, CDR, CDRW).

Obnova dát v MySQL

Teraz vieme, že záložné súbory, či už od úplnej alebo inkrementálnej zálohy sú vlastne iba súbory s bežnými SQL príkazmi. To znamená, že ich môžeme jednoducho použiť pre obnovu dát.

Obnovu dát vykonáme tak, že tieto súbory použijeme v spojení s monitorom *mysql* v dávkovom režime.

Ako prvý použijeme súbor z úplnej zálohy. Ten nech sa volá **kniha.sql**. Syntaktický zápis je:

mysql -u meno_používateľa -p heslo_používateľa názov_databáze < meno_zálohového_súboru

Poznámka: Príslušný používateľ musí mať samozrejme riadne definované práva k tomuto úkonu!

Takže príkazom:

mysql -u root -pheslo kniznica < kniha.sql

sa načíta súbor **kniha.sql** a všetky SQL príkazy, ktoré obsahuje sa riadne vykonajú. (Obrátený zobáčik < má opačný význam ako >, teda činnosť programu sa nebude do súboru **kniha.sql** zapisovať, ale sa obsah súboru bude do programu *mysql* načítavať.

Potom vezmeme jednotlivé súbory inkrementálnych záloh a obdobným spôsobom vykonáme doplnenie rozdielov k dnešnému dňu:

mysql -u root -pheslo kniznica < update.001
mysql -u root -pheslo kniznica < update.002

Cvičný príklad zálohovania a obnovy

Aby sme si príslušnú teóriu precvičili, urobíme si tento cvičný príklad:

1) **Spustíme server** s prepínačom `--log-update`, aby dokázal vykonávať inkrementálne zálohy:

- v linuxe: **safe_mysqld --log-update &**

- vo Windows9X: **mysqld --log-update**

V adresári, kde sa nachádzajú dátové súbory vznikol súbor **mior.001** (*mior* preto, lebo aj server sa volá *mior*).

2) **Majme databázu test** a v nej tabuľku **zaner** s touto štruktúrou a dátami:

CIS_ODD	TEMATIKA
1	poézia
2	román
3	krimi
4	detská lit.
5	cestopis
6	lit. faktú
7	odborná lit.

cis_odd je typu integer int(3) a *tematika* je typu varchar(20).

3) **Urobíme úplnú zálohu** príkazom:

mysqldump --opt -u root -pheslo test zaner >zaner.sql

Obsah súboru **zaner.sql** je na výpise č.III-4-5:

```
# MySQL-Front Dump 2.2
#
# Host: localhost   Database: test
#-----
# Server version 4.0.1-alpha

USE test;

#
# Table structure for table 'zaner'
#

DROP TABLE IF EXISTS zaner;
CREATE TABLE `zaner` (
  `cis_odd` int(3) NOT NULL default '0',
  `tematika` varchar(20) default NULL,
  PRIMARY KEY (`cis_odd`)
) TYPE=MyISAM PACK_KEYS=1;

#
# Dumping data for table 'zaner'
#
INSERT INTO zaner VALUES("1","poezia");
INSERT INTO zaner VALUES("2","roman");
INSERT INTO zaner VALUES("3","krimi");
INSERT INTO zaner VALUES("4","detska lit.");
INSERT INTO zaner VALUES("5","cestopis");
INSERT INTO zaner VALUES("6","lit. faktú");
INSERT INTO zaner VALUES("7","odborna lit.");
```

4) **Zmažeme 5. oddelenie (5.položku)**

Môžeme to urobiť v monitore *mysql* (nezabudnite nastaviť správnu databázu pomocou *use test;*):

```
mysql> delete from zaner where cis_odd = 5;
```

Odpoveď by mala byť:

Query OK, 1 row affected

Opustíme monitor *mysql* príkazom *exit*.

Poznámka: môžeme to urobiť aj v inej ľubovoľnej aplikácii!

Presvedčíme sa, že je piata položka naozaj zmazaná!

5) **Urobíme inkrementačnú zálohu**

Spustíme príkaz v OS: **mysqldadmin -u root -pheslo flush-logs**

Súbor **mior.001** sa naplnil príslušným obsahom a vytvoril sa nový súbor **mior.002**

*(Poznámka: V niektorých windowsovských verziách sa môže vytvoriť len súbor **mior** bez prípony. V linuxe to funguje korektne!)*

Pozrieme sa do súboru **mior.001** - výpis č.III-4-6:

```
# C:\MYSQL\BIN\MYSQLD~1.EXE, Version: 4.0.1-alpha-log at 020721 17:59:32
use test;
delete from zaner where cis_odd = 5;
```

Vidíme, že sa skutočne protokovali naše činnosti. A práve túto vlastnosť teraz využijeme. Dosiahli sme status quo!

6) **Tabuľku zaner zmažeme** - nasimulujeme stratu dát napr. v *mysql* monitore:

```
mysql> delete from zaner;
```

Odpoveď: *Query OK, 6 rows affected*

KRACH!!!

Skontrolujeme, že sa tabuľka **zaner** naozaj vypráznila. Prípadne ju môžeme aj úplne zmazať!

Teraz sme totálne v kaši. Šéf zúri, používatelia nadávajú, vedúca učtárne je na infarkt a riaditeľ nám vypisuje prepúšťací lístok. Čo teraz?

No predsa prístupíme k obnove dát! Zálohy máme, úplnú (**zaner.sql**) aj prírastkovú (**mior.001**), nemáme sa čoho báť!

7) **Obnovíme úplnú zálohu** zo súboru **zaner.sql** príkazom:

```
mysql -u root -pheslo kniznica < zaner.sql
```

Teraz by sa mala vytvoriť tabuľka **zaner** a naplniť sa príslušnými dátami. Presvečíme sa!

8) **Obnovíme inkrementačnú zálohu** zo súboru **mior.001** príkazom:

mysql -u root -pheslo kniznica < mior.001

9) **Skontrolujeme tabuľku**, či je taká, ako bola tesne pred krachom.

Je to naozaj tak! Sme za vodou!

Používatelia sú spokojní, šéf nás uznanlivo potľapkáva po pleci a riaditeľ vypisuje príkaz k odmene. Pani účtovníčka nám naleje koňaku čo jej doniesla dcéra z Paríža a dá aj sebe, vraj je to dobré proti infarktu. A jeden „deň - blbec“ je za nami!

Občas sa stáva, že sa data naozaj porušia a z nevysvetliteľných príčin nemáme ani zálohy. Čo potom? Existuje možnosť opravy dát špeciálnym programom. Ale o tom nabadúce.

Malé veľké databázy III / 5.časť

Minule sme si vysvetlili základy zálohovania a obnovy dát. Túto činnosť budeme spravidla vykonávať pri krachu databázy v tom význame, že došlo k strate alebo značnému porušeniu dát.

Poškodenie databázy však nemusí vždy znamenať jej obnovu z vytvorených záloh. Pokiaľ došlo k poškodeniu dát výpadkom elektrického prúdu alebo „zaseknutím“ serveru, je možné najprv použiť k obnove dát určité nástroje, ktorými systém MySQL disponuje. Dnes sa pozrieme na tieto prostriedky.

Datová štruktúra MySQL

Ešte pred tým, než začneme používať opravné prostriedky MySQL, si musíme povedať niečo o štruktúre dát.

MySQL používa niekoľko formátov tabuliek. Tie sa delia na **netranzakčno-bezpečné** (not transaction-safe tables = NTST) a **tranzakčno-bezpečné** (transaction-safe tables = TST). Medzi NTST patria tieto formáty:

- Ø ISAM
- Ø MyISAM
- Ø HEAP
- Ø MERGE

a medzi TST patria:

- Ø InnoDB
- Ø BDB

Nebudeme si teraz vysvetľovať, ktorý formát na čo slúži, jeho výhody a nevýhody si povieme inokedy. Nám stačí vedieť, že MySQL používala do verzie 3.22 formát ISAM a od verzie 3.23 formát MyISAM.

My sa budeme zaoberať formátom MyISAM, pretože ten sa používa ako štandard pri každej novej inštalácii MySQL.

Poznámka:

Ak by sme z určitých dôvodov potrebovali zmeniť formát už používaných tabuliek, stačí použiť mocný príkaz ALTER TABLE. Jednotlivé formáty sú „nepozerateľné“ v ľubovoľných editoroch, preto ich nebudeme zbytočne otvárať.

V nami používanom operačnom systéme nájdeme adresár, ktorého názov zodpovedá menu databázy, ktorú máme v MySQL definovanú. Nech je to v našom prípade databáza **KNIZNICA**. Ak sa dobre pozrieme na výpis obsahu tohoto adresára, vidíme tieto súbory:

- **kniha.frm**
- **kniha.MYD**
- **kniha.MYI**
- **zaner.frm**
- **zaner.MYD**
- **zaner.MYI**

Je zrejmé, že súbory s názvom **kniha** budú prislúchať tabuľke **KNIHA** a tie ostatné tabuľke **ZANER**. Význam jednotlivých súborov konkretizujú prípony.

Prípona **.frm** je formátovací súbor, ktorý popisuje dátovú štruktúru príslušnej tabuľky.

Prípona **.MYD** obsahuje jednotlivé dáta, teda obsah záznamov tabuľky.

Prípona **.MYI** obsahuje informácie o indexoch a kľúčoch a iných vnútorných krížových odkazoch, prislúchajúcich k danej tabuľke.

(Súbory formátu ISAM majú prípony **.frm**, **.isd** a **.ism**.)

Práve posledné dva súbory je nutné pri poškodení opravovať.

Na opravu môžeme použiť niekoľko prostriedkov:

- Ø CHECK TABLE
- Ø REPAIR TABLE
- Ø OPTIMIZE TABLE
- Ø myisamchk

Posledne uvedený je najuniverzálnejší prostriedok, ktorého parametrizáciou nahradíme účinok tých predchádzajúcich. Preto sa mu budeme teraz venovať. Len pre doplnenie, pre tabuľky typu ISAM sa používa program **isamchk**, ktorého parametre a účinky sú totožné s **myisamchk**.

Použitie príkazu myisamchk

Pred použitím príkazu musíme byť v príslušnom adresári, teda adresár **KNIZNICA** musí byť naším pracovným adresárom. V opačnom prípade musíme uviesť úplnú cestu k vybranej tabuľke.

Obečný zápis príkazu je:

`myisamchk [parametre] meno_tabuľky`

Výber najčastejšie používaných parametrov je v tabuľke III-5-1:

Tabuľka III-5-1: Parametre príkazu myisamchk

Parametre	Zápis	Význam
-a	--analyze	Analyzuje distribúciu kľúčov, je vhodný pre urýchlenie niektorých spojení tabuliek
-d	--description	Vypíše užitočné informácie o tabuľke
-e	--extend-check	Dôkladne preverí vybraný súbor. Používa sa, keď všetko zlyhá
-f	--force	Prepisuje dočasné súbory
-i	--information	Vypíše štatistické údaje o danej tabuľke
-k počet	--keys-used počet	Používa sa s prepínačom -r. Prikazuje programu, aby pred vykonaním opravy deaktivoval kľúče, ktorých je nad stanovený počet. Teda ak k=0, odstráni (deaktivuje) všetky kľúče
-q	--quick	Používa sa s prepínačom -r. Vykoná rýchlu opravu iba indexových súborov.
-r	--recover	Spustí opravu. Opraví väčšinu problémov okrem konfliktov jedinečných kľúčov.
-o	--safe_recover	Používa staršiu metódu obnovy, ktorá je pomalejšia. Dokáže však opraviť aj to, čo nevie parameter -r
-s	--silent	Vypíše iba zistené chyby
-v	--verbose	Režim s podrobným výpisom
-V	--version	Vypíše číslo verzie programu a ukončí sa

Asi najzákladnejším parametrom je parameter **d**.

Použijeme:

`myisamchk -d kniha`

Výpis vyššie uvedeného príkazu je na výpise č. III-5-2:

MyISAM file:	kniha			
Record format:	Packed			
Character set:	latin1 (8)			
Data records:	10	Deleted blocks:	0	
Recordlength:	136			
table description:				
Key	Start	Len	Index	Type
1	2	4	unique	long

Tento výpis okrem iného oznamuje, že tabuľka kniha obsahuje 10 záznamov. Pre nás je v tejto chvíli zaujímavá hodnota u *Delete blocks*. Tá hovorí, koľko zmazaných blokov tabuľka obsahuje. Čím je číslo vyššie, tým viac plýtvame miestom. Prečo?

Ak totiž zmažeme niekoľko záznamov z tabuľky, v skutočnosti neuvoľnía svoje miesto novým. Toto miesto zostáva obsadené, a tým sa zväčšuje aj veľkosť súboru. Pokiaľ sa jedná o nízku hodnotu *Delete blocks*, môžeme zostať pokojní. Ak však hodnota dosiahne určitú vysokú hodnotu, musíme nevyužívané miesto uvoľniť.

Preto použijeme príkaz:

```
myisamchk -r kniha
```

a dostaneme výsledok, podobný výpisu č.III-5-3:

```
- recovering (with sort) MyISAM-table 'kniha'
Data records: 10
- Fixing index 1
```

Vyššie uvedený príkaz prehľadá tabuľku **KNIHA** a na základe zistených informácií ju vytvorí znova, avšak tentokrát bez nevyužívaného miesta. Tým sa zmenší aj veľkosť súboru *kniha.MYD*.

Príkaz *myisamchk -d* by sme mali spúšťať pravidelne, zvlášť v prípadoch, ak používame veľmi „pulzujúcu“ databázu, teda ak sa v nej často používa príkaz *DELETE*. Práve po ňom zostávajú v súboroch prázdne miesta. Len tak môžeme bdieť nad obsadeným miestom na pevnom disku.

Zapamätajte si!

Ak sa pri vykonávaní príkazu myisamchk pokúsí niektorý z klientov pripojiť k databáze, nadobudne program myisamchk mylného dojmu, že sú tabuľky poškodené, aj keď tomu tak v skutočnosti nie je.

Preto pred spustením príkazu myisamchk ukončíme démona (službu) mysqld. (Ak sme si úplne istí, že počas obnovy nebude nikto pristupovať k tabuľkám, stačí, že pred spustením zmieneneho príkazu spustíme príkaz mysqladmin flush-tables.).

Postup pri oprave poškodených tabuliek

Počas vývoja MySQL sa ustálil určitý postup, ktorý napomáha oprave poškodených záznamov. Postupuje sa pritom od najjednoduchšieho k zložitejšiemu.

Ešte pred samotnou opravou je doporučené vykonať zálohu databáze (pomocou *mysqldump*, ak to ovšem ide!).

Keďže z dôvodu poškodenia sa „vydumpovanie“ databáze nemusí podariť, vykonáme zálohovanie fyzických súborov, teda v našom prípade *kniha.frm*, *kniha.MYD* a *kniha.MYI*. To preto, že nie vždy je zaručené, že ten ktorý postup (t.j. parameter) vedie ku kladnému výsledku opravy.

Rýchle opravy

Rýchle opravy vykonávame príkazom:

```
myisamchk -rq názov_tabuľky
```

Teda príkaz:

```
myisamchk -rq kniha
```

a jeho výpis č.III-5-4:

```
- check key delete-chain
- check record delete-chain
- recovering (with sort) MyISAM-table 'kniha'
Data records: 10
- Fixing index 1
```

overí stav tabuľky a v prípade potreby ju opraví. Tento typ opravy sa označuje ako rýchly preto, lebo sa opravajú iba indexové súbory.

Bežné opravy

Väčšinu bežných problémov, okrem konfliktu jedinečných kľúčov opraví príkaz **myisamchk -r**.

Rozsiahlejšie opravy

Na rozsiahlejšie opravy použijeme príkaz **myisamchk -e**. Ten overí stav súborov **.MYI** a **.MYD** a starostlivo hľadá zdroje možného poškodenia súboru. Opraví väčšinu problémov, ale ak sa stretne so závažnou chybou, preruší opravy a ukončí sa.

Príkaz **myisamchk -ev** sa od predchádzajúceho líši nielen podrobnejším výpisom, ale po nájdení závažnej chyby pokračuje v odstraňovaní problému. Ak je to nutné, zmaže konfliktné dáta. Preto pred použitím týchto parametrov vykonáme bezpodmienečne zálohu súborov.

Poznámka:

Používajte príkaz s parametrami v tu uvedenom poradí. Postupným prehlbovaním testov môžeme zistiť skutočný rozsah poškodenia.

Opravy poškodených kľúčov

Cez to všetko, že sú kľúče (indexy) zvyčajne určené ku zvýšeniu výkonu databáze, môže sa stať, že môžu tento výkon naopak znižovať. To platí hlavne u príkazov **INSERT** alebo **UPDATE**. Ak spustíme program **myisamchk** na tabuľku s poškodenými kľúčami, program môže predpokladať, že sú poškodené aj dáta a môže ich vymazať. Ak sa domnievame, že sú poškodené kľúče, dočasne ich odstránime, obnovíme tabuľku a na záver kľúče dosadíme späť.

Príkaz:

myisamchk -rqk 0 (nula nie písmeno O)

vynuluje kľúče a zkontroluje a prípadne zreparuje tabuľku.

Ak budeme s výsledkom opravy spokojní, obnovíme kľúče príkazom **myisamchk -rq**.

Premiestnenie databáze

Pre premiestnenie databáze môžeme mať mnoho dobrých dôvodov. Môžeme napríklad inovovať hardvér alebo aj softvér, alebo chceme jednoducho preniesť databázu na iný počítač. Môžeme mať dokonca kópiu svojej databáze, uloženú na bezpečnom mieste, ktorú potom kopírujeme na www server.

Premiestneniu databáze sa hovorí **replikácia** (replication). Väčšina dobrých veľkých databáz s replikáciou počíta a má ju v sebe integrovanú.

My sa teraz pozrieme, ako by sme to vyriešili v MySQL.

Nech už je dôvod replikácie akýkoľvek, v podstate sa tento proces delí na tri základné kroky:

- Ø uloženie (archivácia)
- Ø premiestnenie
- Ø obnova

My už vieme vykonať archiváciu databáz pomocou príkazu **mysqldump**. Je ale možný ešte jeden spôsob, a to priamo archivovať súbory na disku.

Prenos alebo premiestnenie archivovaných dát alebo súborov závisí na použitom médiu, či už sa jedná o páskové zariadenia, cédečká alebo využitie počítačovej siete.

Obnova dát je závislá od spôsobu archivácie a prenosu.

Musíme si uvedomiť, že využitie fyzických súborov s príponami **.frm**, **.MYD** a **.MYI** môže viesť k nekonzistencii dát a preto sa využíva iba výnimočne.

Z celkového popisu vieme, že to asi nie je tá pravá replikácia. Chcelo by to trochu automatizácie a hlavne zrýchlenie celej činnosti tak, aby replikovaný server bol „on line“.

Preto MySQL zaviedla „one way“ replikáciu od verzie 3.23.15, kde jeden server je *master* a druhý *slave*

A dokonca vo verzii MySQL 4.0 je to ešte zdokonalené.

Ale o tom nabadúce.

Malé veľké databázy III / 6.časť

Dnes sa budeme venovať málo používanej, ale zato veľmi efektívnej činnosti MySQL servera - replikácii. Tak ako sme si spomenuli v predchádzajúcich dvoch častiach, replikácia tesne súvisí s bezpečnosťou - ochranou a obnovou dát. MySQL sa dopracovala do štádia, keď dokáže sama vykonávať tzv. one - way replikáciu. Ale ešte predtým, ako budeme naše databáze replikovať, povieme si niečo o binárnom logovaní.

Binárny update log

Spomínate si, ako sme robili (pevne verím, že od tej doby aj všetci stále robíme) inkrementálne zálohy? Spustili sme server s parametrom **--log-update**. Tým sa do logovacieho súboru zaznamenávali všetky zmeny v našich dátach. Po „vyflušnutí“ sme tieto súbory odkladali na horšie časy.

MySQL server zavádza nový typ logovania zmien, a to v binárnej forme.

Binárny update log, ako by sme toto mohli nazývať, obsahuje všetky informácie ako v klasickom *update logu*, ale v podstatne výkonejšej forme. Okrem iného obsahuje informácie o tom, ako dlho každý dopyt, upravujúci dáta v tabuľke, trval.

Binárny update log bol zavedený hlavne kvôli replikácii.

To, čo sme si povedali a naučili sme sa o činnosti a úlohe textového update logu, platí aj pre tento nový binárny typ.

Aby sme stanovili serveru, že má zapisovať všetky zmeny dát do binárneho update logu, spustíme ho s parametrom **--log-bin**, teda

- v linuxe: **safe_mysqld --log-bin [=meno_súboru] &**

- vo Windows9X: **mysqld --log-bin [=meno_súboru]**

Ak nezadáme žiadne *meno_súboru*, systém sám zvolí meno podľa mena servera a doplní ho príponou **-bin**, na ktorom daný MySQL beží. Ak sa teda môj počítač volá **mior**, systém vytvorí nový binárny logovací súbor s menom **mior-bin.001**. Ak zadáme meno súboru v relatívnej ceste, teda bez presného určenia, kde sa má daný súbor vytvoriť, tak sa tento súbor vytvorí v adresári, kde sú uložené dáta serveru MySQL.

Už sme povedali, že všetky pravidlá pre prácu s logovaním pomocou **--log-update** platia aj pre prácu s **--log-bin**. Takže všetky činnosti okolo rotovania súborov, vyprázdňovania keše a podobne uplatníme aj tu. Preto sa nimi už nebudeme zaoberať.

Okrem súboru **mior-bin.00x**, v ktorom sú zaznamenané vykonané zmeny a ostatné príslušné údaje, systém vytvorí ešte jeden, takzvaný indexový súbor, ktorý sa bude volať **mior-bin.index**. V tomto súbore sú uložené poznámky o vytvorení rotovacích logov. Tento súbor je veľmi dôležitý pri samotnej replikácii.

Zmena nastáva pri obnovovaní dát z uložených logovacích súborov.

Vtedy sme používali príkaz

```
mysql -u root -p heslo kniznica < mior.001
```

To sme mohli použiť preto, lebo súbor **mior.001** bol v textovej forme.

Ak sa však pozrieme do súboru **mior-bin.001**, uvidíme, že to nie je čistý textový súbor a tak nemôžeme na obnovenie dát z tohoto súboru použiť vyššie spomenutý príkaz.

Pre používanie binárnych súborov je MySQL k dispozícii program **mysqlbinlog**. Ak chceme obnoviť dáta z binárneho logu, použijeme príkaz:

```
mysqlbinlog log_súbor | mysql -h meno_servera
```

teda v našom prípade

mysqlbinlog mior-bin.001| mysql -h mior

Môžeme tiež použiť **mysqlbinlog** na čítanie binárnych logov priamo zo vzdialených MySQL súborov.

Ak spustíme príkaz

mysqlbinlog mior-bin.001

dozvieme sa z výpisu na obrazovke obsah súboru, aké zmeny a kedy nastali a ako bol tento súbor zrotovaný - výpis č.III-6-1:

```
# at 4
#020928 9:30:53 server id 1      Start: binlog v 2, server v 4.0.1-alpha-
max-debug-log created 020928 9:30:53
# at 79
#020928 9:42:31 server id 1      Query thread_id=2 exec_time=0
      error_code=0
use test;
SET TIMESTAMP=1033198951;
DELETE FROM articles WHERE id=1;
# at 145
#020928 10:04:47 server id 1      Rotate to mior-bin.002pos=4
# at 184
#020928 10:04:47 server id 1      Stop
```

V prípade, že by sme sa chceli dozvedieť viac o možnostiach samotného príkazu **mysqlbinlog**, spustíme ho s parametrom **--help**.

Replikácia v MySQL

One - way (jednosmerná) replikácia v MySQL je pomerne dobre vypracovaná, robustná a rýchla. Na jej činnosť potrebujeme minimálne dva servery MySQL, teda dva počítače, kde jeden bude **master** (pán) a tie ostatné budú v roli **slave** (sluha), ktoré preberú úlohu mastera, keď na ňom nastanú problémy.

Od verzie 3.23.15, MySQL podporuje one - way replikáciu interne. **Master** server vytvára binárny update log súbor a indexový súbor. **Slave** na základe spojenia s masterom ho informuje, že dohonil zmeny, ktoré prebehli na mastrovi a čaká na ďalšie oznámenie zmien v databázach na mastrovi.

One - way replikácia znamená, že zmeny, ktoré prebehnú na masterovi, budú vykonané aj na slave-ovi. Naopak to však nie je možné. Skutočne sa jedná o replikáciu **iba jedným smerom**!

Princíp činnosti

MySQL one - way replikácia je založená na tom, že *master server* zaznamenáva všetky zmeny v dátach svojich tabuliek a databáz do binárneho logu. Servery typu *slave* čítajú uložené zmeny z tohoto binárneho logu a vykonajú tie isté zmeny na svojich dátach.

Z toho vyplýva, že pred spustením samotnej replikácie je veľmi dôležité mať presnú kópiu dát ako na masteri, tak aj na jednotlivých slavoch. Ak by sme nezabezpečili identické kópie, je pochopiteľné, že by sme na slavoch získavali neidentické údaje!

Rozdielnosť použitých verzií

Ak chceme používať one - way replikáciu, bolo by veľmi dobré, ak by sme mali na oboch systémoch, teda na master-ovi aj slave-ovi rovnakú verziu MySQL. Ak to však z určitých dôvodov nie je možné, platia tieto pravidlá:

Slave verzie 4.0.0 nevie komunikovať s mastrom verzie 3.23.

Slave verzie 4.0.1 a vyššie už dokáže komunikovať s mastrom verzie 3.23.

Slave verzie 3.23 nedokáže komunikovať s masterom verzie 4.0.

Aby sme mohli používať one - way replikáciu, všetky tabuľky musia byť typu MyISAM!

Nastavenie replikácie

Teraz si popíšeme postup, ako nastavíme master-a a slave-a na výkon one - way replikácie:

- 1) Presvedčíme sa, či máme vhodné verzie MySQL na oboch stranách. Doporučuje sa použiť verzie od 3.23.29 a vyššie. Staršie verzie mali chyby v binárnych logoch a nefungovali korektne.
- 2) V systéme MySQL na *masterovi* vytvoríme nového používateľa, ktorý bude mať privilégiá **FILE** a možnosť pristupovať z požadovaných počítačov, ktoré budú zastávať úlohu *slave*. Najvhodnejšie je vytvoriť takého používateľa, ktorý bude vykonávať iba replikáciu, aby nemal povolené iné možnosti v MySQL. Pre ukážku vytvoríme používateľa s menom **repl**, s heslom „karol“, ktorý môže pristupovať z rôznych počítačov typu *slave*. Príkaz na vytvorenie potom bude:

```
GRANT FILE ON *.* TO repl@'%' IDENTIFIED BY 'karol';
```

- 3) Vypneme server MySQL na strane *master*:

```
mysqldadmin -u root -p shutdown
```

- 4) Vytvoríme kópie všetkých tabuliek a databáz na strane *master*. Ak používame na *masterovi* a *slaveovi* tú istú verziu MySQL, môžeme jednoducho preniesť systémové súbory MySQL. Ak nie, použijeme na prenos spôsob popísaný pri archivovaní dát.
- 5) V súbore **my.cnf** na strane *master* pridáme do sekcie **[mysqld]** tieto dve položky:

```
[mysqld]
log-bin
server-id=1
```

Položka **server-id=1** je veľmi dôležitá! Je to unikátne číslo, ktoré identifikuje “adresu” *mastera*, podobne ako IP adresa.

- 6) Reštartujeme server na strane *master*.
- 7) Na strane *slave* pridáme do súboru **my.cnf** tieto položky:

```
master-host=<meno_mastera>
master-user=<meno_replikačného_používateľa>
master-password=<heslo_replikačného_používateľa>
master-port=<TCP/IP port mastera>
server-id=<unikátne číslo medzi 2 až 2^32 - 1>
```

Za uvedené hodnoty dosadíme požadované údaje, teda za *meno_mastera* dáme **mior**, za *meno_replikačného_používateľa* dosadíme **repl**, za *heslo_replikačného_používateľa* dosadíme **karol**. *TCP/IP port mastera* býva 3306, ak sme ho nezmenili.

server-id na strane *slave* musí byť odlišné od *server-id* na strane *master*!

Ak zabudneme nastaviť *server-id* na strane *slave*, dostaneme takúto chybovú hlášku v súbore **error.log**:

```
Warning: one should set server_id to a non-0 value if master_host is set.
The server will not act as a slave.
```

Ak toto zabudneme urobiť na strane *master*, počítače typu *slave* sa nebudú schopné pripojiť na *mastra*. Len čo sa bude *slave* replikovať, nájdeme súbor s názvom **master.info** v tom istom adresári, ako **error.log**. Nemažme ani needitujeme tento súbor!

- 8) Na strane *slave* vytvoríme identickú kópiu celého databázového stroja. Môžeme využiť prenesené súbory zo strany *mastera*.
 - 9) Reštartujeme *slave*.
- Potom, čo to vykonáme, *slave* sa pokúsi spojiť s *masterom* a zachytiť všetky zmeny, ktoré od tej doby nastanú.

Ďalšie parametre súboru **my.cnf**

Vykonávanie samotnej replikácie môžeme ovplyvniť nastavením konkrétnych parametrov. Tie sa spravidla zapisujú do súboru **my.cnf**. Tabuľka č. III-6-2 popisuje najdôležitejšie parametre a ich význam:

Tabuľka č.III-6-2: Parametre súboru my.cnf:

Parameter	Popis
--log-bin=meno_súboru	Systém štandardne vytvorí meno súboru z názvu servera a prípony -bin. Tento súbor sa uloží do adresára, kde sú uložené aj dáta servera MySQL. Ak požadujeme iné uloženie tohto súboru, musíme napísať úplnú cestu, napr.: log-bin = /mysql/logs/replikacia.log. Príponu .log systém pri zrotovaní odstráni a nahradí známym trojčísлом 001, 002 a podobne.
--log-bin-index=meno_index_súboru	Podobne môžeme stanoviť meno indexového súboru.
--binlog-do-db=meno_databáze	Týmto parametrom určíme, ktorá databáza sa má replikovať. Ostatné databázy budú ignorované. Veľmi často toto použijeme, ak chceme replikovať iba konkrétnu databázu, a nie celý databázový stroj. Príklad: binlog-do-db=kniha
--binlog-ignore-db=meno_databáze	Týmto parametrom určíme, ktorá databáza sa nemá replikovať. Ostatné databázy sa replikovať budú. Príklad: binlog-ignore-db=zaner.
--master-port=číslo_portu	Ak sme zmenili štandardný port, na ktorom beží celý databázový stroj MySQL (čo je 3306), tu stanovíme nové číslo portu.

To by sme mali k replikácii všetko.

Naša vysoká škola databáz sa blíži k záveru. Ostáva nám prebrať už len niekoľko posledných kapitol. Povieme si niečo o vstavaných funkciách a pozrieme sa súhrne na najdôležitejšie funkcie systému MySQL.

Ale to už nabudúce.

Malé veľké databázy III / 7. – záverečná časť

V dnešnej, skoro poslednej časti seriálu sa pozrieme na MySQL databázu zo systémového hľadiska. Povieme si niečo o tom, čo by mala silná databáza mať a čo MySQL má či nemá, ako to riešiť a ako vyladiť server na plný výkon.

Výkonný produkt áno či nie?

V dnešných dobách relatívne neobmedzených informácií nastáva potreba čo najvýkonnejšieho databázového produktu. Výkonnosť každého produktu je daná dvomi aspektmi:

- a) robustnosťou samotného databázového systému
- b) optimalizáciou behu systému

Robustnosť systému

Pod týmto pojmom si môžeme predstaviť počet a efektivitu funkcií systému. Systému, ktorý obsahuje viac (naozaj) potrebných funkcií hovoríme, že je **robustnejší**.

Medzi najdôležitejšie funkcie systému zaradíme:

- Ø transakcie - **transactions**
- Ø uložené procedúry – **stored procedures**
- Ø kurzory - **cursors**
- Ø spúšťače - **triggers**
- Ø obmedzujúce pravidlá – **constrains**

Robustnosť je však na úkor rýchlosti systému. Spravidla systém, ktorý obsahuje viac funkcií, a je teda robustnejší, býva aj pomalší.

Naopak, systém, ktorý neobsahuje niektoré funkcie býva veľmi rýchly.

Nesmieme si však zamieňať pojem „rýchly“ s pojmom „výkonný“!

Výkonnosť je akýsi kompromis medzi robustnosťou a rýchlosťou systému.

Pozrime sa informatívne na jednotlivé funkcie a porovnajme ich s možnosťami MySQL.

Tranzakcie

Tranzakcia je skupina spracovávaných činností, ktoré sú kompletne celé dokončené, alebo keď dokončené nie sú, sú databázy a systém spracovávaná ponechané v rovnakom stave ako pred zahájením transakcie. Tranzakcie sú považované za atómicke operácie.

Pre názornosť si uveďme príklad:

Majme databázu **FINANCIE** a v nej tabuľku **VYPLATY** s dvoma stĺpcami **ZAMESTNANEC** a **PLAT**. Teraz nezáleží na type jednotlivých stĺpcov.

Predstavme si, že zrazu nastane fantastický deň, keď sa vedenie podniku rozhodne, že sa všetkým zamestnancom zvyšujú platy o 30%. My ako programátori či obsluha systému musíme zabezpečiť túto úlohu. Zadáme príkaz, aby sa prepočítali všetky platy. Úloha beží, keď tu zrazu nastane neočakávaný výpadok servera! Po obnove servera dostávame nepravdivé informácie. Časť platov je navýšená, zvyšok ešte nie. Ale ktoré sú to?

Práve transakcie zabezpečia, že dáta, nad ktorými úloha nie je z rôznych dôvodov úplne dokončená, sa vrátia do pôvodného stavu.

Preto našu úlohu budeme riešiť pomocou transakcie. V prípade podobného výpadku servera sa dáta vrátia do stavu pred spustením úlohy (teda nezmení sa ani jeden plat, pokiaľ nie sú zmenené všetky). V prípade, že nenastanú žiadne komplikácie, úloha sa správa ako bežná činnosť servera.

Tranzakčná činnosť je vnútorná činnosť databázového stroja, preto sa o ňu veľmi nemusíme starať. Stačí, ak príkazy našej úlohy uzavrieme medzi transakčné príkazy **BEGIN TRANS**, **COMMIT** alebo **ROLLBACK TRANS**.

MySQL už v posledných verziách podporuje transakcie, ale iba na tabuľkách typu InnoDB. V bežne používanom type MyISAM transakcie nefungujú. Treba priznať, že sú iba v začiatkoch a ich kvalita sa bude zlepšovať.

Musíme si uvedomiť, že práve transakcie najviac zdržujú činnosť databázového stroja, teda ho podstatne spomaľujú.

Uložené procedúry

Uložená procedúra je postupnosť preložených (skompilovaných) príkazov SQL (presnejšie jazyka Transact SQL) uložených v databáze priamo na databázovom serveri.

Uložené procedúry ponúkajú mnoho výhod. Jednou z nich je možnosť predávania parametrov. Zároveň zvyšujú bezpečnosť databázy – používateľ, ktorý nemá explicitné práva používať určitú tabuľku, môže s touto tabuľkou pracovať práve pomocou uložených procedúr.

Jedným z hlavných prínosov uložených procedúr je, že sú vykonávané omnoho rýchlejšie ako bežné príkazy. Používanie uložených procedúr nemá v podstate žiadne nevýhody (okrem tej, že takéto procedúry musia byť pripravené dopredu a predom uložené na serveri).

MySQL zatiaľ nemá možnosť uložených procedúr, dá sa však očakávať, že sa v najbližších verziách objavia (presnejšie by mali byť vo verzii 4.1).

MySQL server zatiaľ rieši tento nedostatok metódou tzv. používateľsky definovaných funkcií – *user defined function* – UDF. Sú to funkcie, ktoré musia byť napísané v jazyku C alebo C++.

Kurzory

Kurzory umožňujú pracovať s jednotlivými riadkami výslednej sady. Ich použitie sa veľmi podobá prechádzaniu výslednej sady pomocou objektov rozhrania API. Kurzor umožňuje programátorovi alebo správcovi databázy vykonávať na strane servera pomerne zložité operácie. Je to celkom praktická funkcia, ale práca s kurzormi vyžaduje hlbšie znalosti.

Nevýhodou kurzorov je ich slabý výkon a sú značne pomalé.

MySQL neobsahuje kurzory – a to je len dobre.

Spúšťače

Spúšťač - *trigger* je v podstate uložená procedúra, ktorá sa spustí automaticky ako reakcia na určitú akciu a ktorá slúži k zachovaniu integrity databázy. Akcie, ktoré môžu spustiť spúšťač, sú príkazy *UPDATE*, *DELETE* a *INSERT*.

Majme tabuľky **KNIHY** a **ARCHIV_KNIH**. Predpokladajme, že chceme, aby pri každom odstránení záznamu z tabuľky **KNIHY** sa tento záznam uložil do tabuľky **ARCHIV_KNIH**.

Namiesto toho, aby sme pridávali zložitý kód do aplikácie, ktorá záznam odstráni a presunie do druhej tabuľky, alebo aby sme volali uloženú procedúru ručne, môžeme použiť spúšťač. Ten sa automaticky spustí pri každom odstránení záznamu z tabuľky **KNIHY** a jeho úlohou je odstraňovaný záznam uložiť do tabuľky **ARCHIV_KNIH**.

Spúšťače majú celú radu výhod. Uľahčujú zachovávanie referenčnej integrity a využívajú sa k zaisteniu platnosti relácií medzi záznamami zviazaných tabuliek. Práve preto, že sú spúšťané automaticky ako dôsledok inej akcie, môžu slúžiť ako nástroj automatickej údržby databázy.

Ďalšou zaujímavou vlastnosťou spúšťačov je kaskádový efekt. K tomu dochádza, keď spúšťač spustený ako dôsledok určitej udalosti v jednej tabuľke vyvolá spustenie iného spúšťača, ktorý zase reaguje na príslušnú udalosť vo svojej tabuľke. To môže byť na jednej strane veľmi užitočné, ale na strane druhej aj veľmi nebezpečné.

Majme tabuľky **OBCHODY**, kde vedíme obchodnú činnosť, **OBJEDNÁVKY**, kde evidujeme objednávky jednotlivých zákazníkov a **OPERÁCIE**, kde sú evidované všetky obchodné operácie v súvislosti na objednávkach. Predpokladajme, že chceme odstrániť všetky záznamy v tabuľke **OBCHODY**, ktoré sa viažu ku konkrétnemu zákazníkovi. Namiesto toho, aby sme tvorili špeciálny program v našej aplikácii, ktorý by po danom zákazníkovi „upratal“, použijeme spúšťač. Vytvoríme taký spúšťač, ktorý odstráni všetky záznamy v tabuľke **OBJEDNÁVKY** viazané na odstraňovaného zákazníka. A k tabuľke **OBJEDNÁVKY** môžeme pripojiť ďalší spúšťač, ktorý odstráni všetky obchodné operácie spojené s odstraňovanými záznamami o objednávkach v tabuľke **OPERÁCIE**.

Uvedomme si, že spúšťače sú uložené na strane servera, ich kód je v pamäti systému a spúšťajú sa automaticky! Preto ich vôbec nemusíme volať v zdrojovom kóde našej aplikácie!

Nevýhodou spúšťačov je fakt, že spomaľujú a zaťažujú systém. Pri každej operácii nad danou tabuľkou sa musí databázový stroj presvedčiť, či sa na danú akciu neviaže niektorý spúšťač. Ak áno, musí ho spustiť a vykonať. Táto činnosť blokuje čas procesora a tým čas celého systému.

MySQL nemá možnosť použitia spúšťačov. Môžeme to však obísť dobrým návrhom databázy a následných aplikácií. Najlepšie je zaisťovať integritu databázy priamo v kóde aplikácií. Upratujeme po sebe v dátach a nebudeme potrebovať spúšťače!

(Aj keď musím priznať, že by som takúto funkciu v MySQL privítal. V prípade, že by som ju nechcel používať, dal by som to serveru pri spustení najavo nejakým prepínačom, aby sa značne zrýchlil. A keď by som ich v inej aplikácii potreboval, tak by som ich zase „zapol“).

Podľa vyjadrenia autorov MySQL, triggermi sa budú zaoberať od verzie 4.1 a vyššie.

Obmedzujúce pravidlá

Obmedzujúce pravidlo (*constraint*) je spôsob, ako vynútiť dodržiavanie pravidiel zaistenia platnosti relácií medzi záznamami zviazaných tabuliek. Existuje niekoľko typov obmedzujúcich pravidiel, ale všetky majú spoločný cieľ – zaistenie referenčnej integrity.

Predstavme si pole tabuľky alfanumerického typu, ktoré by malo obsahovať iba písmená. Môžeme nastaviť obmedzujúce pravidlo, ktoré používateľovi zabráni vložiť akékoľvek číslo! A tento kód nemusíme implementovať na strane aplikácie, ale priamo na serveri.

Obmedzujúce pravidlá však veľmi zaťažujú systém a preto sa v MySQL nenachádzajú. Zaistenie integrity dát je teda plne v rukách programátorov a správcov databáze.

Optimalizácia behu systému

Veľmi často (teda skoro **vždy**!) sa stáva, že po určitej dobe funkčnosti nášho dobrého projektu opadne počiatočné nadšenie a programátori a používatelia sa začnú sťažovať na pomalosť celého systému. Preto bude potrebné pristúpiť k optimalizácii systému, čím dosiahneme určité zrýchlenie.

Faktorov, spomaľujúcich samotnú rýchlosť systému je niekoľko a spravidla vždy sa v zmysle Murphyho zákonov spájajú do tej najhoršej kombinácie.

Činnosti, eliminujúce tieto faktory, si môžeme rozdeliť do týchto skupín:

- Ø dolad'ovanie výkonu databáze
- Ø zostavovanie lepších dopytov SQL
- Ø uvoľňovanie nevyužitého priestoru
- Ø úprava a kompilácia zdrojových kódov servera

Dolad'ovanie výkonu databáze

Jedným z prvých faktorov, na ktorý by sme sa mali zamerať, je samotný systém. Na akom type počítača je spustený náš databázový server? Koľko pamäti má k dispozícii? Aký rýchly je procesor? A disk?

Samozrejme že najlepšie by bolo, keby bol databázový server spustený na samostatnom stroji najmodernejšej konštrukcie s čo najlepšími parametrami. Ale sami vieme, že to nie je vždy možné a preto treba pristúpiť k rozumným kompromisom.

Našťastie, MySQL (na rozdiel od iných databázových strojov) nie je veľmi náročná na hardvér. Asi najviac ho bude „tlačiť“ veľkosť operačnej pamäte. To preto, lebo MySQL je viacvláknový systém. Pri nadviazaní spojenia s klientom je vytvorené nové vlákno – podproces, ktorého úlohou je plniť požiadavky klienta. Každé vlákno požaduje určitú časť operačnej pamäte. Takže čím viac pamäte, tým lepšie! Systém s väčšou operačnou pamäťou býva výkonnejší.

Preto, ak musíte voliť medzi rýchlejšim procesorom alebo väčšou pamäťou, siahnite po pamäti!

Ďalšia oblasť, ktorá ovplyvňuje výkon databázového servera je pevný disk. Rýchlejší disk zaručuje rýchlejšie výsledky. Najoptimálnejšie by bolo uložiť tabuľky na samostatný disk.

Asi najpodstatnejším faktorom v tejto téme je použitý operačný systém. MySQL je vypracovaná apriori pre systém Linux a tak na tomto systéme dosahuje podstatne lepšie výsledky, ako keby sme ho používali v systéme MS Windows. Vyplýva to z koncepcie prideľovania pamäte jednotlivým procesom samotným operačným systémom.

Systémové premenné

Ak máme najlepší hardvér, alebo už lepší nezískame, mali by sme sa trochu pohrať so samotným databázovým systémom. Výkon systému riadi mnoho systémových premenných.

Ak zadáme príkaz:

mysqladmin -p variables

po zadaní rootovského hesla získame výpis systémových premených, ktoré sú prednastavené pri behu samotného démona (služby) *mysqld*.

Jednotlivé premenné môžeme zmeniť a tým značne vyladiť beh servera.

Taktiež zadaním parametrov pri spustení *mysqld* dosiahneme zlepšeného behu systému.

Zostavovanie lepších dopytov SQL

Ďalším boľavým miestom pomalosti systému sú príkazy SQL, ktoré pristupujú k dátam. Preto pri návrhu databáze budeme dodržiavať tieto najzákladnejšie pokyny:

- Ø Používajme čo najmenšie dátové typy. Čím menší dátový typ, tým menej miesta potrebuje na disku, ale aj v pamäti.
- Ø Stĺpce by nemali obsahovať prázdne hodnoty.
- Ø Vyhýbajme sa poliam premennej dĺžky.
- Ø Nepoužívajme priveľa indexov. (Preštudujme si kapitoly o indexoch!).
- Ø Typ tabuľky určujeme vzhľadom na jej budúci výkon. Táto schopnosť sa dá dosiahnuť iba praxou.
- Ø Používajme implicitné hodnoty.
- Ø Zostavujme dopyty tak, aby v maximálnej miere používali indexy.
- Ø Používajme vo výpisoch výsledkov kľúčové slovo *LIMIT*.
- Ø Venujme pozornosť klauzule *WHERE*.

Na tému *WHERE* by sa dala napísať celá kniha (aj sú naozaj napísané!). Verte, dobre vytvoriť správny dopyt je naozaj umenie!

Uvoľňovanie nevyužitého priestoru

Po zmazaní niektorého záznamu v tabuľke zostáva tzv. hluché miesto. Toto sa v súbore tabuliek kumuluje a tým zaberá zbytočne miesto na disku. Aby sme odstránili tieto hluché miesta, použijeme príkaz *myisamchk*, ktorého použitie sme si už vysvetľovali nedávno.

Preklad a kompilácia zdrojových kódov servera

Väčšina z nás na začiatku svojej práce s týmto produktom využila pri inštalácii predkompilované súbory, či už vo verzii tgz, rpm alebo exe. Tieto predkompilované súbory sú nastavené tak, aby fungovali pre čo najširší záber úloh.

Ak však chceme podstatne vylepšiť činnosť MySQL, doporučujem preložiť server zo zdrojových kódov vrátane nastavenia príslušných parametrov ešte pred samotnou kompiláciou.

V prípade Linuxu to nie je až taký problém, zatiaľ čo neviem o nikom, čo by kompiloval verziu pre MS Windows.

Záver. Záver?

Toto je **skoro posledná** časť seriálu *Malé veľké databázy*. **Posledná** preto, lebo seriál v tejto podobe naozaj končí. A **skoro** preto, lebo sa predsa len občas stretneme – na základe vašich mailov a otázok občas pripravím malú prednášku o rôznych fintách a novinkách tejto vynikajúcej databáze.

Ohliadnutie späť

Pamätáte sa na to, keď sme začínali?

Nemali sme o databázach žiadne vedomosti. Prešli sme úplnými základmi (1.séria), vytvorili sme prvé funkčné aplikácie, ktoré boli „oknoidné“, či už sme použili MS Excel, www server Apache a PHP, alebo sme ich naprogramovali v Borland Delphi a povedali sme si zásady správneho návrhu aplikácií (2.séria).

V 3.sérii sme si povedali niečo o indexoch, kľúčoch a zámkoch. Venovali sme sa bezpečnosti, archivácií dát a ich obnove, o opravách poškodených tabuliek a o replikácii a dnes o vyladení systému.

Promócie

Prešli sme školou databáz – od základnej, cez strednú až po vysokú. A preto tí, ktorí prešli a zvládli celý tento seriál, získavajú titul **DBDr. – DataBase Doctor**, teda doktor databázológie. Otvorme šampanské, vypustíme svetlice, vystískajme kámošky!

Ale nezaspíme na vavrínoch!

A nakoniec len pre zaujímavosť:
Vedzte, že tento seriál obsahuje:

- Ø 165 knižných strán
- Ø 8546 riadkov
- Ø 56 088 slov
- Ø 372 276 znakov
- Ø 194 obrázkov, výpisov alebo grafov
- Ø v celkom 26 kapitolách.

A to je, myslím, pomerne dosť.

Nech sa nám darí vo svete databáz!

Miroslav Oravec

P.S:

Mám rád databáze všeobecne, mám rád túto báječnú databázu a mal som rád aj tento seriál. Asi mi bude za ním smutno...

Ale vraj sa hovorí, že kde jedno končí, tam iné začína. Takže hádam predsa len – niekedy dovidenia!