Implementace SQL 2 Intermediate Level a SQL 3 / PSM ve WinBase602 5.1 a použité optimalizační techniky

Januš Drózd, Software602 a.s., Pod Višňovkou 25, 140 00 Praha 4, Česká republika

Abstrakt

Příspěvek se zabývá implementací jazyka SQL 2 na úrovni Intermediate v databázovém serveru WinBase602 a jazyka SQL 3 / PSM (*persistent stored modules* - procedury a funkce uložené na serveru a prováděné serverem). Popisuje techniky optimalizace SQL dotazů se zvláštním zaměřením na optimalizaci predikátu OR a vybrané novinky připravované normy SQL 3.

Mezi nimi je popsán zejména způsob ošetřování výjimek při běhu procedur uložených na serveru.

1. Vývoj norem SQL a současný stav

Normalizace v oblasti spolupráce relačních databází a jejich klientských aplikací má za cíl dosáhnout stavu, kdy databázové aplikační programy nebudou svázány s konkrétním databázovým serverem, ale budou schopny (nejlépe bez úprav) využívat služeb libovolného serveru splňujícího požadavky určité úrovně nějakého všeobecně přijatého standardu. Takovýto cílový stav je nanejvýš žádoucí pro vývojáře i pro uživatele aplikací. Mezi dodavateli serverů budou z normalizace profitovat spíše menší a střední firmy. Pro společnosti dominující na trhu serverů je výhodnější si vytvářet vlastní ohraničené nekompatibilní impérium, z něhož zákazníci nemohou zběhnout ke konkurenci.

Normalizace spolupráce klienta s databázovým serverem zahrnuje kromě normalizace syntaxe a sémantiky jazyka SQL také otázky včleňování komunikace s databází do prostředí hostitelských programovacích jazyků. Zejména se pak normalizuje programové rozhraní, dovolující volat služby serveru pomocí sady funkcí - Call Level Interface.

Normalizace vycházející z iniciativy organizací ANSI a ISO přinesla dosud:
- normy SQL-86 a SQL-89, dnes všeobecně akceptované výrobci serverů,

- normu SQL-92 označovanou také SQL 2, s níž se dodavatelé serverů v současnosti potýkají,
- řadu pracovních návrhů normy SQL 3.

Norma SQL 2 definuje tři úrovně, na nichž může být implementována v konkrétním produktu: Entry Level, Intermadiate Level a Full Level. Vícero výrobců deklaruje, že plně implementovalo Entry Level, a většina výrobců zahrnuje do svých produktů také vybrané rysy z vyšších úrovní. Mimo oficiální normy existuje navíc tzv. Transient Level, ležící mezi Entry a Intermediate Level.

Norma SQL 3 zatím nemá definitivní podobu, nicméně některé její části (například jazyk pro psaní procedur uložených na serveru, tzv. PSM) se již prakticky nevyvíjejí. Hlavní přínos SQL 3 by měl spočívat v oblasti objektově orientovaných databází, v ní lze však ještě očekávat zásadní změny normy.

Úsilí o normalizaci SQL trpí typickými chorobami tvorby norem v oblasti softwarových technologií. Autoři norem občas podlehnou pokušení zahrnout do normy vše, co jim připadá užitečné nebo elegantní. Co hůře, někdy se pokusí prostřednictvím normy předepsat budoucí vývoj oboru. Na rozdíl od pejska a kočičky, kteří upekli nepoživatelný dort, produkty podle takovýchto norem neupeče obvykle nikdo. Pro vývojáře je totiž bližší konkrétní zájem uživatele databáze než vize normotvůrce.

Norma SQL 2 přináší například implementačně poměrně obtížné *assertions*, tedy globální pravidla, jejichž zachovávání má server vynucovat. Nezahrnuje však *triggery*, jednoduché a funkční nástroje, které lze k prosazení globálních pravidel využít. Procedury uložené na serveru jsou v normě SQL 2 tak omezeny, že ztrácejí smysl.

Norma SQL 3 vyniká mimo jiné svým rozsahem. Skládá se ze 7 částí, z nichž základní část *Foundation* má cca 1200 stran. Další části jsou psány formou změn základní části. Pokud chcete zjistit, co platí o určité syntaktické konstrukci, musíte vyjít z textu ve *Foundation* a aplikovat na něj změny (nahrazení, přidání a vypuštění odstavců) popsané v dalších částech.

Mimo toto „veřejnoprávní" normalizační úsilí existují také firemní standardy, které se deklarují jako „otevřené". Z nichž je nejznámějším ODBC (Open Database Connectivity) firmy Microsoft. Vývoj těchto norem je určován momentální firemní strategií a některé jejich podivné zákruty se dají vysvětlit potřebami aktuální fáze boje s konkurencí. Aktuální verze ODBC 3.0 se v řadě míst odvolává na SQL 2.

2. Úroveň SQL implementovaná ve WinBase602 5.1

Verze jazyka SQL implementovaná ve WinBase602 5.1 si bere za základ SQL 2, Intermediate Level. Od této úrovně se však na řadě míst odchyluje, ve velké většině případů směrem vzhůru.

Odchylky od Intermediate Level směrem dolů se týkají zejména těch rysů, v nichž se rozchází koncepce normy s dlouhodobou koncepcí WinBase. Například Intermediate Level stanovuje, že kromě typu TIME musí existovat i typ TIME WITH TIME ZONE, tedy čas obsahující údaj o časové zóně. Takový typ ve WinBase602 neexistuje.

WinBase602 zahrnuje vybrané rysy SQL 2 Full Level. Z nich je zřejmě nejpodstatnější možnost používat dotazový výraz uzavřený v závorkách ve funkci tabulky, tedy například v klauzuli FROM.

Z SQL 3 je do WinBase zahrnuta prakticky kompletní část PSM, tedy jazyk pro tvorbu procedur uložených na serveru a triggerů. Mimo tuto část je z SQL 3 implementováno pouze minimum rysů, například typy BOOLEAN, BLOB a CLOB (které překvapivě v SQL 2 chybí).

3. Optimalizace dotazů ve WinBase602 5.1

Zpracování klientských příkazů systémem řízení báze dat probíhá ve dvou fázích. V první, přípravné fázi se generuje na základě analýzy příkazu a databázového schématu plán zpracování příkazu, v druhé, výkonné fázi se tento plán provádí.

Důvodem rozdělení zpracování příkazu do dvou fází je možnost ušetření množství času, pokud se provádění stejného nebo téměř stejného příkazu mnohokrát opakuje. Pak stačí opakovat výkonnou fázi, zatímco přípravná fáze se provede pouze jednou. V některých způsobech napojení aplikace na databázový server může být přípravná fáze provedena již při kompilaci klientského programu nebo skrytě při jeho spuštění.

Přípravná fáze zpracování začíná syntaktickou analýzou příkazu, neboť příkazy se typicky předávají v textové podobě. Převedením textového řetězce do interních struktur databázového serveru tato fáze pro řadu příkazů skončí. Výjimkou jsou příkazy obsahující dotazový výraz, ať už explicitní (otevření kurzoru, příkaz SELECT INTO), nebo skrytý v podmínkách tvořících součást příkazů (příkazy UPDATE ... WHERE či DELETE ... WHERE). V nich přípravná fáze pokračuje optimalizací dotazu, jejímž výsledkem je plán postupu vyhodnocování dotazu.

Optimalizace dotazů je nejzajímavější částí zpracování SQL příkazů. Většina výrobců databázových serverů považuje techniky optimalizace za průmyslové tajemství a nepublikuje je. Místo toho se k uživatelům dostávají zprávy, že produkt používá novou technologii s exotickým jménem vymyšleným marketingovými specialisty. Jelikož předpokládaný čtenář je asi vnímavější na racionální než emocionální argumenty, tento článek poodhalí způsob optimalizace použitý ve WinBase602.

3.1. *Interní reprezentace syntaxe dotazu*

Gramatika SQL, publikovaná v normě, není jednoznačná, což by bylo možno považovat za důkaz ignorance autorů normy vzhledem k teorii formálních jazyků. Naštěstí se dá poměrně jednoduše převést na jednoznačnou gramatiku některého snadno analyzovatelného typu. Ve WinBase602 jsem použili LL(2) gramatiku, z níž lze jako ze všech LL gramatik triviálně odvodit analyzátor pracující metodou rekurzivního sestupu. Převod gramatiky do LL(1) podoby by výrazně zvýšil její složitost.

Syntaxe dotazu v SQL v sobě nese pečeť dlouhé historie tohoto jazyka. Operace relační algebry, které by logicky měly stát vedle sebe, se zapisují principiálně odlišnými syntaktickými konstrukcemi. Prvním úkolem analýzy je proto převést zápis dotazu do struktur, které adekvátně zachycují operace relační algebry vyjádřené dotazem. Použili jsme stromovou strukturu, v níž listy odpovídají persistentním tabulkám z databázového schématu, a vnitřní uzly reprezentují tyto (binární a unární) operace:

A) sjednocení, rozdíl a průnik tabulek;
B) vnitřní join;
C) některý druh vnějšího joinu;
D) výběr řádek splňujících určité podmínky;
E) rozdělení řádek do skupin a splynutí každé skupiny do jedné řádky pomocí agregačních funkcí.

Každý z těchto uzlů může navíc zahrnovat operaci, v níž se vytvoří nová množina sloupců tak, že hodnota v každém novém sloupci definuje jako funkce hodnot ve starých sloupcích.

Takto vzniklý strom nazveme *stromem struktury dotazu*. Reprezentuje dotaz v pojmech relační algebry a odhlíží od vyjadřovacích specifik jazyka SQL.

3.2. *Samostatně optimalizované podstromy*

Optimalizace prováděná na stromu struktury dotazu jako celku se jeví jako příliš složitý úkol. Proto je tento strom rozdělen do podstromů, které se optimalizují samostatně. Při vhodném návrhu komunikace mezi algoritmy optimalizujícími podstromy nemusí toto rozdělení přinést žádné zhoršení výsledku optimalizace.

Rozdělení stromu syntaxe do podstromů je nejmenší relace ekvivalence obsahující relaci R definovanou takto:

<*x*, *y*> patří do R když *x* je otcem *y* ve stromu syntaxe a x je typu D;

<*x*, *y*> patří do R když *x* je předkem *y*, *x* i *y* jsou typu B a na cestě z *x* do *y* jsou pouze uzly typů B a D.

Takto vzniklé třídy ekvivalence jsou uzly nového stromu, který nazveme stromem optimalizace. Listy jsou stejně jako ve stromu struktury dotazu persistentními tabulkami.

Vnitřní uzly obsahují vždy množinu omezení na řádky, převzatou z uzlů typu D a z podmínek v joinech, a jednu z těchto operací:

- sjednocení, rozdíl a průnik dvou tabulek;
- vnitřní join dvou nebo více tabulek;
- některý druh vnějšího joinu dvou tabulek;
- rozdělení řádek do skupin a splynutí každé skupiny do jedné řádky pomocí agregačních funkcí

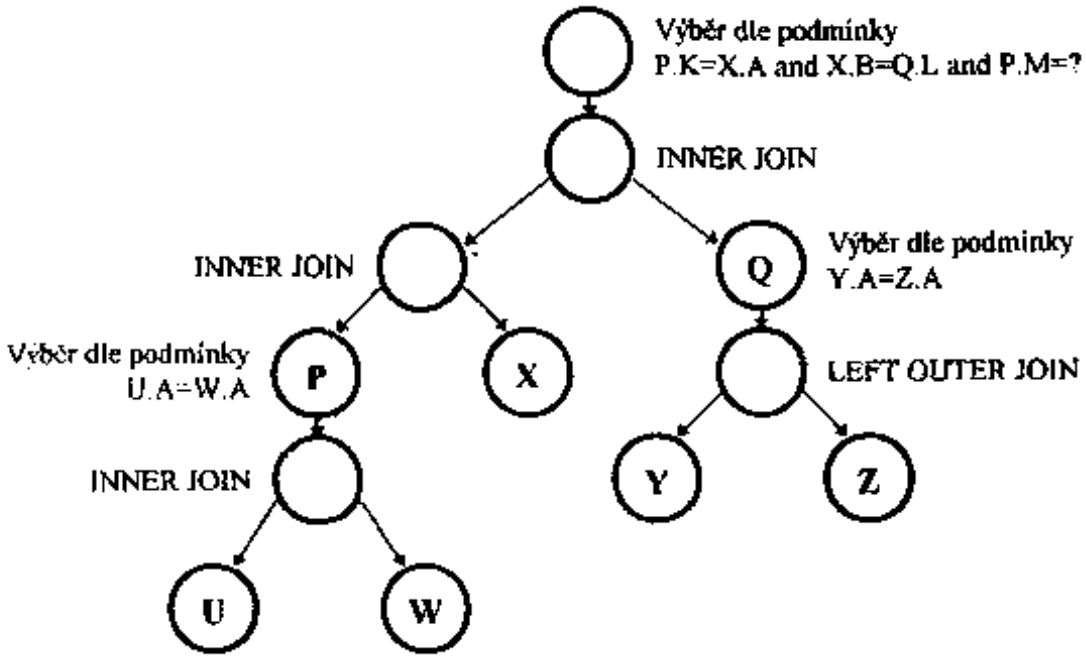
3.3. *Příklad*

Mějme persistentní tabulky U, W, X, Y, Z jejich sloupce nechť jsou A a B. Schéma obsahuje tyto deklarace:
create view P **as select** U.B+W.B **as** K, W.B **as** M **from** U, W **where** U.A=W.A
create view Q **as select** Y.B **as** L **from** Y **left outer join** Z **on** (Y.A=Z.A)

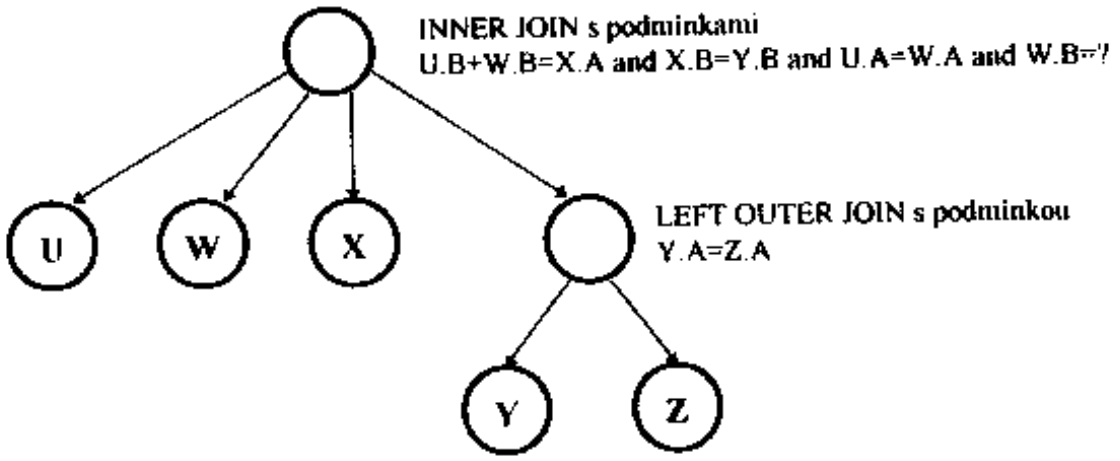
Optimalizujeme dotaz:
select * **from** P, X, Q **where** P.K=X.A **and** X.B = Q.L **and** P.M=?

Znak ? označuje dynamický parametr. Při optimalizaci dotazu jeho hodnotu neznáme, ale budeme ji znát při vyhodnocování.

Strom struktury dotazu vypadá takto:



Strom optimalizace vypadá takto:



Strom optimalizace obsahuje dva vnitřní uzly, v nich bude probíhat optimalizace.

3.4. Optimalizace vnitřního uzlu ve stromu optimalizace

Uvažujme nejprve optimalizaci jednoho vnitřního uzlu a zanedbejme jeho okolí. Chceme najít co nejlepší způsob vyhodnocení operace relační algebry předepsané v uzlu, s tím, že z výsledku nás zajímá pouze část vyhovující podmínce P. Nejzajímavější je případ optimalizace n-árního vnitřního joinu.

Klasický postup spočívá v tom, že hledáme pořadí propojování tabulek T_1 až T_n. Podmínku P pak přepíšeme do tvaru P_1 and P_2 and ... P_n tak, že P_x obsahuje odkazy pouze na sloupce tabulek T_1 až T_x (to lze provést vždy, v krajním případě P_1 až P_{n-1} jsou TRUE a P_n je P). Výsledek relační operace pak sestrojíme tak, že:

- projdeme záznamy tabulky T_1 a vybereme ty, které vyhovují podmínce P_1, tak získáme mezivýsledek M_1;
- pro každé i od 2 do n sestrojíme mezivýsledek M_i jako join M_{i-1} a T_i zúžený podmínkou P_i;
- M_n je výsledkem operace.

Optimalizace spočívá v tom, že hledáme takové pořadí tabulek, v němž je sestrojení výsledku nejrychlejší. Doba vyhodnocení bude tím kratší, s čím menším počtem záznamů budeme pracovat, tedy čím větší zúžení množiny záznamů přinese každá podmínka P_i.

Pro malá n lze projít všechny permutace tabulek a najít nejlepší z nich (starší verze SQL serveru WinBase602 takto postupovaly). Pro vyšší aritu joinu roste faktoriál příliš rychle a je nutný jiný postup. Přestavme si graf, v něm vrcholy jsou všechny posloupnosti nad některými z tabulek T_1 až T_n a hrana vede z uzlu X do uzlu Y tehdy, když Y je prodloužením X o jednu tabulku. V takovémto grafu můžeme hledat nejkratší cestu z uzlu

odpovídajícího prázdné posloupnosti do kteréhokoli uzlu reprezentujícího nějakou posloupnost všech n tabulek. WinBase602 používá Dijkstrův algoritmus nejkratší cesty. Ve výše uvedeném příkladu je možným výsledkem například:
T_1 = W, T_2 = U, T_3 = X, T_4 = výsledek outer joinu
P_1 = W.B=?, P_2 = U.A=W.A, P_3 = X.A=U.B+W.B, P_4 = Y.B=X.B

3.5. *Optimalizace s podmínkami obsahujícími OR a NOT*

Unární operátory NOT odstraníme z podmínky tak, že je necháme podle de Morganových pravidel sestoupit až do elementárních výrazů. Problémem jsou binární operátory OR, které brání v efektivním rozdělení podmínky P do P_1 až P_n.

WinBase602 postupuje tak, že na základě podrobnější analýzy podmínky P zařadí operátory OR do jedné ze tří tříd, které zde charakterizujeme pouze pomocí příkladů:

1) Podmínky ve stylu A=B or A=B+1;
2) Podmínky ve stylu A=B or F(A), kde F je funkce nedovolující vnitřní optimalizaci;
3) Podmínky ve stylu A=... or B=... .

Ve všech příkladech předpokládáme, že A, B a C patří do různých tabulek.
V podmínkách patřících do první uvedené kategorie neznamená operátor OR žádnou komplikaci a při optimalizaci můžeme postupovat stejně, jako kdybychom pracovali s jednou podmínkou A=F(B) nebo B=F(A).

V druhé kategorii se nenabízí žádná viditelná možnost, jak se operátoru OR zbavit.
Třetí kategorie je nejzajímavější a optimalizační techniky vyvinuté pro WinBase602 mohou přinést podstatné zrychlení vyhodnocování dotazu oproti klasickým optimalizačním postupům. Uvažujme například dotaz:
SELECT * FROM X, Y WHERE X.A=Y.A and (X.B=? or Y.B=?)

Optimalizace výše popsaným klasickým postupem by vedla na nutnost projít jednu z tabulek exhaustivně. WinBase602 postupuje takto:

1. Rozdělí podmínku na dvě, a to X.A=Y.A and X.B=? a X.A=Y.A and Y.B=?
2. Provede klasickou optimalizaci dotazu nejprve s jednou a pak s druhou podmínkou.
3. Sestaví výsledný plán vyhodnocení dotazu jako vyhodnocení dvou dotazů a spojení odpovědí.

Místo jednoho dotazu s neefektivním postupem vyhodnocování WinBase602 tedy přejde ke dvěma dotazům, z nichž každý je optimalizovaný *jinak*, a každý se dá efektivně vyhodnotit.

Tento postup přináší velmi podstatné zefektivnění jisté třídy dotazů obsahujících operátor OR. Nejobtížnější je rozhodování, do které třídy operátor patří a zda rozdělení dotazu může přinést prospěch.

3.6. Koordinace algoritmů optimalizujících vnitřní uzly

Cílem této koordinace je dosáhnout toho, aby optimalizace každého uzlu respektovala kontext vytvořený ostatními uzly. Existuje mnoho metod, jak této koordinace dosáhnout, ve WinBase602 jsme využili postup shora dolů a dědění podmínek.

Postup shora dolů znamená, že nejprve se optimalizuje vrchol optimalizačního stromu a pak se optimalizují podstromy tvořené jeho syny. Toto pravidlo se rekurzivně uplatní na všech úrovních.

Dědění podmínek znamená, že když při optimalizaci uzlu U bylo získáno pořadí synů T_1 až T_n a rozklad podmínky na P_1 až P_n, pak se do optimalizace syna T_i předají ty z podmínek P_1 až P_i, v nichž se vyskytují sloupce z T_i.

Ve výše uvedeném příkladu to znamená, že optimalizace left outer joinu tabulek Y a Z zdědí podmínku X.B=Y.B.

4. Jazyk procedur a triggerů

Rozšíření jazyka SQL dovolující vytvářet plnohodnotné programové konstrukce pro zápis procedur uložených na serveru a triggerů, je normováno v SQL 3 / PSM. Tento standard přišel z hlediska potřeb databázového světa pozdě - v době, kdy již všechny významnější servery měly vlastní jazyky. Rozdíly mezi těmito jazyky nejsou zvlášť velké, stačí však na to, aby bránily snadné přenositelnosti aplikací mezi servery. Nejvýznamnější odlišnost jazyků se týká způsobu ošetřování výjimek, a také toho, jaké funkce mohou uložené procedury plnit a jak se jim předávají parametry a získávají od nich výsledky.

WinBase602 se rozhodla implementovat jazyk definovaný normou SQL 3 / PSM - možnou alternativou by bylo přilnutí k některému existujícímu firemnímu standardu, bez nejmenšího vlivu na jeho vývoj.

4.1. Jazyk PSM

Jazyk PSM obsahuje (vedle všech příkazů SQL) tyto řídící příkazy a konstrukce:

- přiřazovací příkaz **SET**:
  SET *cíl_přiřazení* = *výraz*;
- složený příkaz **BEGIN ... END**:
  [*návěští* :] BEGIN [[NOT] ATOMIC] [*deklarace_proměnných* ...]
  [*deklarace_rutiny* ...] [*deklarace_výjimky* ...] [*deklarace_kurzoru* ...]
  [*deklarace_handleru* ...] [*příkaz* ...] END [*návěští*];
- podmíněný příkaz **IF**:
  IF *podmínka* THEN *příkaz* ... [ELSEIF *podmínka* THEN *příkaz* ...] ...
  [ELSE *příkaz* ...] END IF;
- jednoduchý příkaz **CASE**:

CASE *výraz* { WHEN *výraz2* THEN *příkaz*... }... [ELSE *příkaz*...] **END CASE**;

- vyhledávací příkaz **CASE**:
  CASE { WHEN *podmínka* THEN *příkaz*... }... [ELSE *příkaz*...] END CASE;
- příkaz cyklu **LOOP**:
  [*návěští* :] LOOP *příkaz* ... END LOOP [*návěští*];
- příkaz cyklu **WHILE**:
  [*návěští* :] WHILE *podmínka* DO *příkaz* ... END WHILE [*návěští*];
- příkaz cyklu **REPEAT**:
  [*návěští* :] REPEAT *příkaz* ... UNTIL *podmínka* END REPEAT [*návěští*];
- příkaz pro opuštění konstrukce **LEAVE**:
  LEAVE *návěští*;
- příkaz cyklu přes všechny záznamy v kurzoru **FOR**:
  [*návěští* :] FOR *řídicí_proměnná* AS [*jméno_kurzoru*
  [SENSITIVE | INSENSITIVE] CURSOR FOR] *specifikace_kurzoru*
  DO *příkaz*... END FOR [*návěští*];
- příkaz vyvolání výjimky **SIGNAL**:
  SIGNAL *identifikátor_výjimky*;
- příkaz předání výjimky ke zpracování na vyšší úrovni **RESIGNAL**:
  RESIGNAL [*identifikátor_výjimky*];
- příkaz volání procedury **CALL**:
  CALL *jméno* ([*skutečný_parametr* {, *skutečný_parametr* }...]);
- příkaz specifikující hodnotu funkce **RETURN**:
  RETURN *výraz*;

Konstrukce jazyka obsahují párové závorky (např. LOOP ... END LOOP), není tedy nutno pro vložení série příkazů do konstrukce používat dodatečné složené příkazy. Příkaz GOTO neexistuje, je nahrazen pro strukturované programování vhodnějším příkazem LEAVE, v němž návěští udává, kterou ze zanořených konstrukcí má opustit.

Při volání rutin (procedur a funkcí) lze použít poziční i klíčové parametry. Formální parametry lze označit jako vstupní, výstupní nebo vstupně-výstupní. Mohou mít implicitní hodnotu, která se uplatní, není-li při volání rutiny skutečný parametr uveden.

Lokální proměnné (stejně jako všechny ostatní lokální objekty) se deklarují v libovolném složeném příkazu, tedy nikoli v rutině

4.2. Zpracování výjimek

Mechanismy zpracování výjimek slouží programátorovi k tomu, aby mohl popsat, jak se v určité části programu má reagovat na chybové a výjimečné stavy, a nemusel přitom testovat výsledek každého příkazu, který může výjimku vyvolat.

Některé servery používají klauzuli EXCEPTION, která se uvede na konci složeného příkazu a obsahuje příkazy prováděné jako reakce na výjimku před opuštěním konstrukce.

SQL 3 / PSM definuje syntakticky poněkud složitější, ale flexibilnější zápis. Pomocí deklarace výjimky ve tvaru:

DECLARE *identifikátor* CONDITION [FOR SQLSTATE [VALUE] *sqlstate*];
lze deklarovat identifikátor uživatelské výjimky a případně jej sdružit s některým stavem SQL serveru *sqlstate*.

Ošetření výjimky se specifikuje v deklaraci handleru ve tvaru:

DECLARE { CONTINUE | EXIT | REDO | UNDO } HANDLER
FOR *výjimka* {, *výjimka* }... *příkaz*;

v němž se s výskytem *výjimky* sdruží provedení *příkazu* a přidají se doplňující informace o jedné ze 4 variant dalšího postupu:

- CONTINUE říká, že (po provedení příkazu) se pokračuje za místem, v němž k výjimce došlo;
- EXIT říká se opustí složený příkaz, v niž je handler deklarován, a pokračuje se za ním;
- REDO říká, že se odvolají všechny změny v obsahu databáze, provedené ve složeném příkazu, v němž je handler deklarován, a složený příkaz se začne provádět od začátku znovu;
- UNDO říká, že se odvolají všechny změny v obsahu databáze, provedené ve složeném příkazu, v němž je handler deklarován, a pokračuje se za ním;

Handlery typu REDO a UNDO se smějí vyskytnout pouze ve složeném příkazu označeném jako ATOMIC. Takovýto složený příkaz se z hlediska obsahu databáze provádí jako jediná, nedělitelná akce, a tudiž může být také jako celek odvolán. Ve WinBase602 jsou složené příkazy ATOMIC implementovány pomocí implicitního SAVEPOINTu na začátku příkazu. Provedení REDO nebo UNDO handleru vyvolá ROLLBACK TO SAVEPOINT, normální opuštění složeného příkazu tento savepoint zruší.

Způsob použití výjimek demonstrujeme na příkladu průchodu všech záznamů v kurzoru bez použití cyklu FOR. Modifikujeme tabulku Ceník tak, že v položkách ceníku začínajících na X snížíme cenu o 10%, v položkách začínajících na A zvýšíme cenu o 100% a ostatní položky smažeme.

```
PROCEDURE ZmenaCeniku();
BEGIN
 DECLARE err_notfound BIT DEFAULT FALSE;
 DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'
   BEGIN SET err_notfound=TRUE; END;
 DECLARE curcen SENSITIVE CURSOR FOR
  SELECT cislo_pol, cena FROM Cenik FOR UPDATE;
 DECLARE Polozka CHAR(20);
 DECLARE Cena NUMERIC(14,2);
 OPEN curcen;
```

```
 LabelLoop: LOOP
   FETCH NEXT FROM curcen INTO Polozka, Cena;
   IF err_notfound IS TRUE THEN LEAVE LabelLoop; END IF;
   IF SUBSTRING(Polozka FROM 1 FOR 1) = "X" THEN
    UPDATE SET Cena = Cena*0.9 WHERE CURRENT OF curcen;
   ELSEIF SUBSTRING(Polozka FROM 1 FOR 1) = "A" THEN
    UPDATE SET Cena = Cena*2 WHERE CURRENT OF curcen;
   ELSE
    DELETE WHERE CURRENT OF curcen;
   END IF;
 END LOOP LabelLoop;
 CLOSE curcen;
END;
```

5. Závěr

Přes početné výhrady k normám SQL nezbývá než ocenit jejich přínos k přenositelnosti aplikací a nahraditelnosti serverů. Toho využívá WinBase602, když vývojářům nabízí nástroje pro vývoj široce kompatibilních databázových aplikací.