Databáze standardu SQL, díl 18.

Předposlední díl našeho seriálu ze světa databází standardu SQL je zde a s ním i návod, jak pracovat s procedurami. Ale nepředbíhejme. Nejdříve samozřejmě dokončíme problematiku z dílu minulého.

Jak uložit proceduru

Od okamžiku, kdy jsme studovaný systém opustili naposledy, je už provedením předchozích operací dobře nastaven a kdokoli v něm bude manipulovat s daty, neporuší jeho integritu. Tak vznikl reálně životaschopný systém, kde ubrání jakéhokoli integritního omezení způsobí chybnou funkci. Přidání dalších omezení je plýtváním. Následující tabulka 1 ukazuje počet integritních omezení tabulek podle jejich typu.

Zůstaňme ještě chvíli v DDL a pokusme se vytvořit hierarchický systém příslušných pohledů. První VIEW UDANICKO zobrazí pohled do všech tří tabulek současně:

```
CREATE VIEW UDANICKO(RCU, JMENOU, PRIJMENIU, KOEU, RCO, JMENOO, PRIJMENIO,
KOEO, DEN, CIC, NAZEVC, PRACHY, FINAL) AS

SELECT RCU,U.JMENO,U.PRIJMENI, U.KOEUD,RCO,O.JMENO,
O.PRIJMENI,O.KOEO,DEN,CIC, NAZEVC,CENAC,CENAC*U.KOEOD*O.KOEOB FROM CIN, UDANI,
CLOVEK U, CLOVEK O

WHERE CIC=CICINU AND RCU=U.RC AND RCO=O.RC;
```

Virtuální tabulka obsahuje opravdu všechno potřebné a nahrazuje tabulku UDANI. Další tři VIEW se hodí k sumárnímu pohledu na udavače, jejich oběti a přečiny:

```
CREATE VIEW UDAVAC(RC, JMENO, PRIJMENI, KOEFICIENT, POCET, CELKEM, UPRAVENO)

AS SELECT RCU, JMENOU, PRIJMENIU, KOEU, COUNT(*), SUM(PRACHY),SUM(FINAL)

FROM UDANICKO GROUP BY RCU;


CREATE VIEW OBET(RC, JMENO, PRIJMENI, KOEFICIENT, POCET, CELKEM, UPRAVENO)
```

```
AS SELECT RCO, JMENOO, PRIJMENIO, KOEO, COUNT(*), SUM(PRACHY),SUM(FINAL)
FROM UDANICKO GROUP BY RCO;

CREATE VIEW PRECIN(CISLO, NAZEV, CENA, POCET, CELKEM,UPRAVENO)
AS SELECT CIC, NAZEVC, PRACHY, COUNT(*), SUM(PRACHY),SUM(FINAL)
FROM UDANICKO GROUP BY CIC;
```

Pod vlivem minulých dílů seriálu snadno vytvoříme pohledy na rodná čísla udavačů a obětí:

```
CREATE VIEW RCUDAV(RC) AS SELECT RCU FROM UDANI GROUP BY RCU;

CREATE VIEW RCOBET(RC) AS SELECT RCO FROM UDANI GROUP BY RCO;
```

Poslední VIEW je pro vás malým rébusem:

```
CREATE VIEW MEDAILE(RC, JMENO, PRIJMENI) AS
SELECT RC, JMENO, PRIJMENI FROM OBET
WHERE RC NOT IN RCUDAV;
```

Pro završení trpkého humoru se vraťme do DML a zadejme několik příkazů SELECT, které prověří důkladnost předchozí přípravy:

```
SELECT TOP 10 PERCENT * FROM UDAVAC ORDER BY POCET DESC;
SELECT * FROM UDANICKO WHERE RCU IN RCOBET OR RCO IN RCUDAV;
SELECT COUNT(*) POCET_KUSU FROM MEDAILE;
```

Co ještě zbývá

Umíme už pracovat s tabulkami, které obsahují data vzájemně provázaná doménovými, entitními a referenčními integritami, a jsme schopni zajistit bezrozpornost uložených dat pomocí definic tabulek

v DDL. Dále víme, jak se efektivně podívat do jedné nebo více tabulek pomoci VIEW. Zatím však nevíme, jak efektivně pracovat s příkazy pro aktualizaci dat v tabulkách. Ideální by bylo mít možnost formulovat jeden příkaz, který na serveru vyvolá spuštění jednoho nebo více příkazů. Naštěstí jazyk SQL DDL takové řešení přímo nabízí pomocí uložených procedur. Ty mají svůj název a vnitřní obsah tvořen jednotlivými příkazy. Uloženou proceduru je možné vytvořit, zrušit a spustit. Po spuštění procedury se vykonají její vnitřní příkazy v předem stanoveném pořadí podle algoritmu v proceduře. Pokud jste již programovali v některém jazyce, nebude pro vás obtížné konstruovat i složitější algoritmy. Na druhé straně právě proto nechávám uložené procedury jako poslední téma. Pokud by čtenář o jejich existenci a možnostech věděl dříve, patrně by nebyl ochoten k dekompozici systému do více tabulek a k vnímání integritních omezení a všechno by chtěl řešit algoritmicky. Chtěl jsem zabránit tomu, aby znalci algoritmizace, ke kterým se také hrdě hlásím, nevyrazili kvapem po slepé koleji tvorby obrovských strukturovaných procedur, a to bez stop databázového myšlení. Přes to všechno by bez uložených procedur nebylo možno realizovat rozumně žádný databázový systém. Dalším kladným rysem uložených procedur je jejich nepřerušitelnost. Proceduru spustíme jedním příkazem a ten se provede naráz, přestože může mít složitou vnitřní strukturu. Vhodná konstrukce procedur vede potom k minimalizaci kolizí s integritními omezeními a záleží pouze na nás, jak tuto možnost využijeme.

Procedura bez parametrů

Každý začátek může být lehký, je-li laťka dostatečně nízko. Nejsnazší je vytvořit uloženou proceduru, která nemá žádné parametry. Typické jsou procedury zajišťující hromadný úklid. Procedura KONCIME postupně zruší data v tabulkách A, B a C. Takovou proceduru vytvoříme v DDL příkazem:

```
CREATE PROCEDURE KONCIME
AS
BEGIN
DELETE FROM A;
DELETE FROM B;
DELETE FROM C;
END
```

Pak proceduru snadno spustíme příkazem:

```
EXECUTE PROCEDURE KONCIME;
```

Výhody takové procedury jsou patrné, pokud tabulky B a C jsou číselníky a tabulka A do nich odkazuje. Pak procedura KONCIME ruší obsahy tabulek ve správném pořadí a nebudou problémy

s referenčními integritami. Dále nemusíme znát názvy původních tabulek podobně jako ve VIEW a do třetice přístupové právo k proceduře KONCIME nemusí mít každý nezodpovědný jedinec, ale například jenom správce databáze.

Procedura se vstupními parametry

Rozšíříme možnosti procedur o komunikaci prostřednictvím vstupních parametrů. Každý vstupní parametr je dán svým jménem a datovým typem. Jejich seznam se při vytváření procedury uvede uzavřený do kulatých závorek za názvem procedury. Vstupní parametry mají sice mnohdy podobné názvy a význam jako jednotlivé sloupce v tabulkách, ale jsou to pouze lokální proměnné, které vznikají uvnitř procedury pro její vnitřní potřebu. Při volání procedury do těchto proměnných zvenku vstupují konkrétní hodnoty, které jsou uvnitř procedury použity jako součásti výrazů. Pro rozlišení názvů sloupců tabulek a názvů lokálních proměnných se používá dvojtečková konvence. Je-li před názvem uvedena dvojtečka, jde o název lokální proměnné. Často potřebujeme proceduru pro rušení osoby podle rodného čísla. Nejprve vytvoříme jednoduchou proceduru:

```
CREATE PROCEDURE KILLER(RCX VARCHAR(10))
AS
BEGIN
DELETE FROM CLOVEK WHERE RC=:RCX;
END
```

Po spuštění procedury KILLER příkazem:

```
EXECUTE PROCEDURE KILLER ”5511273208”;
```

si uvědomíme, že konkrétní osobu není možné zrušit, protože má vazby z jiných tabulek. Proto proceduru nejprve zničíme a vytvoříme dokonalejší dílo zkázy. To vše ovšem za předpokladu, že tabulky A, B, C neobsahují zásadní informace, které je nutno uchovávat i po smrti:

```
DROP PROCEDURE KILLER;

CREATE PROCEDURE KILLER(RCX VARCHAR(10))
AS
```

```
BEGIN
DELETE FROM A WHERE RC=:RCX;
DELETE FROM B WHERE RCIS=:RCX;
DELETE FROM C WHERE RRCCIISS=:RCX;
DELETE FROM CLOVEK WHERE RC=:RCX;
END
```

Procedury se vstupními parametry hrají zásadní roli při aktualizaci dat. Následující procedura je vhodná pro změnu křestního jména konkrétní osoby:

```
CREATE PROCEDURE KRESTNI (RCX VARCHAR(10), NOVE VARCHAR(30))
AS
BEGIN
UPDATE CLOVEK SET JMENO=:NOVE WHERE RC=:RCX;
END
```

K přejmenování konkrétní osoby použijeme příkaz:

```
EXECUTE PROCEDURE KRESTNI ”510611030”, ”JOE”;
```

Konečně můžeme i přidávání nového člověka do tabulky chápat jako proceduru zaštiťující jeden komplikovaný přikaz:

```
CREATE PROCEDURE NOVY_CLOVEK (RCX VARCHAR(10), JX VARCHAR(30), VX INTEGER)
AS
BEGIN
INSERT INTO CLOVEK (RC, JMENO, VYSKA) VALUES (:RCX,:JX,:VX);
END
```

Volání je opět jednoduché:

```
EXECUTE PROCEDURE NOVY_CLOVEK ”6104115471”, ”ANNIE”, 9;
```

Procedury, které něco vracejí

V některých případech potřebujeme, aby procedura vrátila zjištěné hodnoty lokálních proměnných. Při vytváření procedury uvedeme seznam vracených proměnných v závorce za klíčové slovo RETURNS. Následující poněkud umělý, ale názorný příklad procedury NANECO ukazuje, jak lze vytvořit snadno současně druhou a třetí mocninu celého čísla:

```
CREATE PROCEDURE NANECO (X INTEGER) RETURNS (X2 INTEGER, X3 INTEGER)
AS
BEGIN
:X2=:X*:X;
:X3=:X*:X*:X;
END
```

Zajímají-li nás mocniny čísla 7, musíme mít deklarovány dvě proměnné, například P a Q. Potom vyvoláme proceduru příkazem s klíčovým slovem RETURNING_VALUES před výstupními parametry:

```
EXECUTE PROCEDURE NANECO 7 RETURNING_VALUES :P, :Q;
```

Někdy potřebujeme uvnitř procedury spustit příkaz SELECT tak, aby vypočetl důležité údaje z tabulek, a to například pomocí agregačních funkcí. Pokud nechceme jako odpověď tabulku, použijeme v příkazu SELECT klíčové slovo INTO až na konci. Za ním uvedeme seznam lokálních proměnných, do kterých má být uložen výsledek. Následují ukázky použití na procedurách NEJMENSI, KDOTOJE, UCET_TED a STAV_TED:

```
CREATE PROCEDURE NEJMENSI (JJJ VARCHAR(30)) RETURNS (VVV INTEGER)
AS
BEGIN
SELECT MIN(VYSKA) FROM CLOVEK
```

```
    WHERE JMENO=:JJJ
    INTO :VVV;
    END

    CREATE PROCEDURE KDOTOJE (RCX VARCHAR(10)) RETURNS (JJJ VARCHAR(30), PPP
VARCHAR(30))
    AS
    BEGIN
    SELECT JMENO, PRIJMENI FROM CLOVEK
    WHERE RC=:RCX
    INTO :JJJ, :PPP;
    END

    CREATE PROCEDURE UCET_TED (CUC VARCHAR(20)) RETURNS (P DECIMAL(10,2),
V DECIMAL(10,2))
    AS
    BEGIN
    SELECT SUM(CASTKA) FROM UCET WHERE CU=:CUC AND POHYB="P"
    INTO :P;
    SELECT SUM(CASTKA) FROM UCET WHERE CU=:CUC AND POHYB="V"
    INTO :V;
    END

    CREATE PROCEDURE STAV_TED(CUC VARCHAR(20)) RETURNS (STAV DECIMAL(10,2))
    AS
    DECLARE VARIABLE A DECIMAL (10,2);
    DECLARE VARIABLE B DECIMAL (10,2);
    BEGIN
```

```
EXECUTE PROCEDURE UCET_TED :CUC RETURNING_VALUES :A, :B;

:STAV=:A - :B;

END
```

První tři uvedené procedury se hodí na zjištění nejmenší výšky člověka podle křestního jména, na identifikaci člověka z rodného čísla a na sumarizaci příjmů a výdajů na účtu. Poslední procedura pro celkový stav na účtu je zajímavá ve dvou směrech. Předně demonstruje možnost volání procedury procedurou s uložením dílčích výsledků do proměnných A, B. Dále vidíme, jak řešit nedostatek lokálních proměnných. Mezi klíčovými slovy AS a BEGIN jsou deklarovány dvě lokální proměnné A, B, které nejsou ani vstupními, ani výstupními parametry procedury. Zajímá-li nás stav účtu, stačí se z klientu zeptat:

```
EXECUTE PROCEDURE STAV_TED ”6674157-471/0531” RETURNING_ VALUES :ST;
```

Větvení v proceduře

Na předchozích příkladech bylo snadné pochopit princip procedur a předávání parametrů. Pro realizaci užitečnějších procedur budeme muset umět řídit postup výpočtu. Začneme větvením, které používá konstrukce IF-THEN-ELSE k podmíněnému provádění příkazů. Chceme-li přidat osobu do tabulky, u které NEMÁME OMYLEM zajištěnou entitní integritu, stačí napsat přidávací proceduru PRIDEJ_HO:

```
CREATE PROCEDURE PRIDEJ_HO (RCX VARCHAR(10), JJJ VARCHAR (30))

AS

BEGIN

IF NOT EXISTS(SELECT RC FROM CLOVEK WHERE RC=:RCX)

THEN INSERT INTO CLOVEK (RC, JMENO) VALUES (:RCX, :JJJ);

END
```

Pokud neexistuje v tabulce CLOVEK ani jeden řádek se stejným rodným číslem jako RCX, je založena nová položka s tímto rodným číslem a příslušným jménem JJJ. V opačném případě se neděje nic. Vidíte názorně, jak nevýchovné jsou jednoduché příklady. Leckdo si teď pomyslí, že primární klíče a unikátní indexové soubory jsou k ničemu. Hlavní smysl integritních omezení je v tom, že nás nezávisle hlídají například i při spouštění nedomyšlených procedur. Představte si, že v proceduře PRIDEH_HO by omylem chyběla spojka NOT. Inteligentnější procedura PRIDEJ_INFO by mohla mít

stejné parametry, ale jiné chování. V případě již existujícího rodného čísla RCX nebude rezignovat, ale opraví jméno člověka:

```
CREATE PROCEDURE PRIDEJ_INFO (RCX VARCHAR(10), JJJ VARCHAR (30))
AS
BEGIN
IF NOT EXISTS(SELECT RC FROM CLOVEK WHERE RC=:RCX)
THEN INSERT INTO CLOVEK (RC, JMENO) VALUES (:RCX, :JJJ);
ELSE UPDATE CLOVEK SET JMENO=:JJJ WHERE RC=:RCX;
END
```

Při volání procedury PRIDEJ_INFO nemusíme vědět předem, zda jde o nového, či o starého známého. U klientu se pak setře rozdíl mezi opravou a přidáním dat. Pokud uvedenou techniku považujete za hazard, používejte větvení na řešení jiných situací. Pak se vám jistě bude líbit procedura:

```
CREATE PROCEDURE ZRUS_HO (RCX VARCHAR(10))
AS
BEGIN
IF NOT EXISTS(SELECT RC FROM UCET WHERE MAJITEL=:RCX)
THEN DELETE FROM CLOVEK WHERE RC=:RCX;
END
```

Jaromír Kukal

Další možnosti řízení výpočtu

Než se pustíme do dokončení projektu OZNAMKA, rád bych popsal obecné možnosti řízení výpočtu uvnitř procedur. Nejdůležitější je pojem bloku. BLOK je skupina příkazů sevřených mezi BEGIN a END. Uvnitř procedury je vždy jeden blok tvořící její tělo. Bloky se mohou též zahnízďovat a vytvářet tak struktury. Posloupnost příkazů uzavřená do bloku se totiž nejen v SQL chová jako jeden příkaz. Toho využíváme jak při větvení, tak při cyklech. Příkaz větvení má dva obecné tvary:

```
IF logický výraz
```

THEN příkaz provedený nejvýše jednou

IF logický výraz

THEN příkaz provedený při splnění podmínky

ELSE opačný příkaz

Logické výrazy a příkazy již známe dávno a bloky usnadní konverzi posloupnosti příkazů na jeden příkaz.

Pro cyklus s testováním podmínky před započetím práce se používá schéma:

WHILE logický výraz

DO příkaz opakovaný několikrát

Pro cyklus přes všechny řádky tabulky určené selectem se používá schéma:

FOR select příkaz

DO příkaz aplikovaný na řádek selectu

Pro ošetření chyb použijeme schéma:

WHEN ANY

DO příkaz číhající na chybu

Pro předčasný východ z procedury použijeme příkaz EXIT.

Použití cyklu, bloku a ošetření výjimek je uvedeno v následujících dvou procedurách. Procedura GAUSS řeší Gaussův školácký problém sečtení čísel od 1 do N. Procedura PRUSVIH řeší stejný problém, ale hledá nejvyšší možné číslo, pro které se výpočet ještě nezhroutí. Všimněte si práce s bloky a lokálními proměnnými:

```
CREATE PROCEDURE GAUSS (N INTEGER) RETURNS (S INTEGER)
AS
DECLARE VARIABLE K INTEGER;
BEGIN
:K=0;
:S=0;
WHILE :K<=:N
DO BEGIN
:S=:S+:K;
:K=:K+1;
END
END

CREATE PROCEDURE PRUSVIH RETURNS (K INTEGER,S INTEGER)
AS
DECLARE VARIABLE KNEW INTEGER;
DECLARE VARIABLE SNEW INTEGER;
BEGIN
:K=0;
:S=0;
WHILE YES
DO BEGIN
:SNEW=:S+:K;
WHEN ANY
DO EXIT;
:KNEW=:K+1;
:S=:SNEW;
```

```
:K=:KNEW;

END

END
```

Po načerpání znalostí o uložených procedurách se můžeme pustit do dokončení projektu a tím i seriálu v příštím díle.

Jaromír Kukal