Programování komunikace s databází

Tato kapitola pojednává o práci programu s daty uloženými na serveru WinBase602. **Týká se prostředí vnitřního programovacího jazyka a vlastností společných pro všechna programovací prostředí.**

Do této kapitoly nejsou zahrnuty syntaxe vnitřního jazyka ani specifika komunikace s databázovým serverem z externích jazyků. Těmto tématům se věnují zvláštní kapitoly tohoto manuálu.

Jak aplikace pracuje s databází?

WinBase602 je postavena na modelu *klient-server*. To znamená, že každá databázová aplikace, která potřebuje pracovat s daty uloženými v databázi, tak činí prostřednictvím serveru. Pouze server je oprávněn provádět manipulace s daty. Přitom prověřuje přístupová práva klienta k datům, synchronizuje požadavky různých klientů a zaručuje zachování integrity databáze.

Klient žádá server o provedení služby jedním ze dvou odlišných způsobů:

- provádí SQL příkazy: vytvoří příkaz jazyka SQL ve formě textového řetězce a zašle ho serveru pomocí volání funkce **`SQL_execute`** (resp. **`SQL_prepare`** a **`SQL_exec_prepared`**);
- volá specifické funkce **WinBase602**, případně (ve vnitřním jazyce) používá speciální konstrukce pro přístup k databázi.

Mnohé akce (např. vkládání a přepisování záznamů v databázi, přidělování práv) se dají alternativně provést jedním nebo druhým způsobem. V této kapitole se budeme věnovat přístupu specifickému pro **WinBase602**.

Přístup k datům

Klientské rozhraní serveru **WinBase602** nabízí po přístup k datům dvě možnosti:

- buď přímo odkazovat na tabulky, v nichž jsou data fyzicky uložena,
- anebo využít dotazy, které z jedné nebo více tabulek vyberou potřebná data a z nich vytvoří KURZOR - virtuální tabulku s odpovědí na dotaz.

Chcete-li pracovat s údajem v databázi, dostanete se k němu tak, že zadáte:

- **tabulku nebo kurzor,**
- **číslo záznamu (v této tabulce či kurzoru),**
- **sloupec v tomto záznamu (pokud se operace netýká záznamu jako celku).**

Kurzory se skládají, podobně jako tabulky, z posloupnosti záznamů členěných do sloupců. Sloupce mají svá jména (určená v dotazu) a typy.

Druhy kurzorů

Kurzorům, které vznikají jako odpověď na dotaz uložený v databázi, říkáme PEVNÉ KURZORY, protože názvy a typy jejich sloupců jsou pevně dány. Ostatní kurzory jsou PROMĚNNÉ KURZORY, protože jejich struktura je určena dotazem položeným při jejich vytvoření.

Čtení a zápis dat ve vnitřním jazyce

K vyznačení záznamu, s nímž chcete pracovat, připojíte k identifikátoru tabulky nebo kurzoru hranaté závorky obsahující číslo záznamu. Pokud chcete odkázat na určitý sloupec, připojíte tečku a jméno sloupce:

```
tabulka_nebo_kurzor[číslo_záznamu].jméno_sloupce
```

Sloupec databázového záznamu lze v programech použít všude tam, kde lze použít proměnnou (není-li to explicitně popřeno). Lze tedy provádět aritmetické operace nad hodnotami sloupců nebo sloupci hodnotu přiřazovat. Manipulace s databázovými sloupci se přeloží jako volání různých služeb databázového serveru.

Sloupec databázového záznamu nelze použít v případě parametru podprogramu předávaného odkazem. Z tohoto pravidla platí jedna užitečná výjimka: je-li formální parametr typu řetězec znaků, pak skutečným parametrem předávaným odkazem smí být i sloupec typu řetězec znaků. Pokud však do formálního parametru přiřadíte v podprogramu novou hodnotu, hodnota databázového sloupce se tím nezmění.

Tabulky a kurzory

Tabulky

Každá tabulka, s níž chcete pracovat, musí být v programu deklarována (na globální úrovni, tedy nikoli jako lokální v podprogramu) a její identifikátor musí být stejný jako jméno, pod nímž je tabulka uložena v aplikaci.

Tabulka uvedeného jména musí existovat již při překladu programu. Touto cestou nelze proto pracovat s tabulkami, které vzniknou dynamicky za běhu aplikace. Protože neexistují proměnné pro tabulky, každá operace s tabulkou v programu je svázána s jednou konkrétní tabulkou v aplikaci. **Tato omezení nemají proměnné kurzory.**

Výhodou přímé práce s tabulkou je její jednoduchost. Již při překladu programu jsou známy všechny její sloupce a lze je proto v programu používat.

Příklad:

```
TABLE Knihy, `Nové výpůjčky`;
begin
   Knihy[cis_exist].vypůjčena := true;
   Knihy[cis_exist].dat_vyp := Today;
   Knihy[cis_exist].dat_vra := Today+limitDays;
   `Nové výpůjčky`[x].kniha := id;
```

Pevné kurzory

Pevný kurzor má stejné jméno, jako určitý dotaz uložený v aplikaci. Pod tímto jménem se pevný kurzor musí v programu deklarovat na globální úrovni (tedy nikoli jako lokální v podprogramu).

Deklarace pevného kurzoru nemůže obsahovat jméno dotazu, který vznikne dynamicky až při běhu aplikace a tento dotaz nemůže odkazovat na tabulky vznikající dynamicky. Tato omezení nemají proměnné kurzory.

Otevření a zavření kurzoru

Pevný kurzor je nutno před použitím otevřít funkcí **`Open_cursor`**. Po použití se uzavírá funkcí **`Close_cursor`**. Opomenutí otevřít kurzor vede k chybě při běhu programu, opomenutí uzavírat kurzory může vést k extensivnímu čerpání zdrojů na serveru.

Příklad:

```
CURSOR Externisti;
begin
  Open_cursor(Externisti);
  r:=Insert(Externisti);
  Externisti[r].stav:=Nevyužit;
  Close_cursor(Externisti);
```

Proměnné kurzory

Proměnné kurzory vznikají z dotazů sestavených až za běhu programu. Deklarují se jako proměnné jedním z těchto dvou způsobů:

```
VAR identifikátor_kurzoru : CURSOR;
VAR identifikátor_kurzoru : CURSOR typ_záznam;
```

Otevření a zavření kurzoru

Kurzor je nutno před použitím otevřít. Proměnné kurzory se otevírají funkcí **`Open_sql_cursor`** nebo **`Open_sql_parts`** a přitom se zadává dotaz. Zavírají se funkcí **`Close_cursor`**.

Výhodou proměnného kurzoru je flexibilita - může obsahovat odpověď na libovolný dotaz a může pracovat i s dynamicky vzniklými tabulkami. S tím však souvisí odlišný přístup ke sloupcům, protože jejich jména ani typy nejsou známy při překladu programu a mohou se dynamicky měnit.

Typ záznam, který lze uvést v deklaraci proměnného kurzoru, slouží k popisu struktury jeho sloupců. **Uvedením tohoto typu autor programu zaručuje, že počet, pořadí, jména a typu sloupců v odpovědi na dotaz uvedený při otevírání kurzoru jsou stejné jako počet, pořadí, jména a typy složek v typu záznam.** Tím se zjednoduší přístup k hodnotám ve sloupcích a jména složek v typu záznam se budou používat k označení

sloupců v kurzoru. Pro popis multiatributů a sloupců proměnné velikosti slouží v typu záznam slova MULTI a FLEXIBLE (viz syntaxe vnitřního jazyka).

Pokud typ záznam není v deklaraci proměnného kurzoru uveden, je přístup k hodnotám ve sloupcích pomalejší a je omezen na přiřazení.

Záznamy

Záznam v tabulce resp. v kurzoru se v programu určuje pomocí svého čísla.

Záznamy jsou očíslovány od nuly až po číslo o 1 menší, než je počet záznamů v tabulce resp. kurzoru. Číslo záznamu (někdy říkáme ABSOLUTNÍ ČÍSLO ZÁZNAMU pro odlišení od interních čísel záznamů ve formulářích) se uvádí ve všech operacích se záznamy. Číslo záznamu v tabulce je po celou dobu existence tohoto záznamu neměnné, s výjimkou operace zkompaktnění tabulky.

Pro vložení záznamu do tabulky nebo kurzoru slouží funkce **`Insert`** a **`Append`**, pro zrušení záznamu funkce **`Delete`**. Všechny záznamy v tabulce nebo kurzoru lze zrušit najednou pomocí funkce **`Delete_all_records`**.

Zrušený záznam v tabulce lze obnovit pomocí funkce **`Undelete`**. Zrušené záznamy lze uvolnit funkcí **`Free_deleted`**, po jejímž provedení již není možno je obnovit. Funkce pro zkompaktnění tabulky **`Compact_table`** kromě uvolnění zrušených záznamů přeuspořádá záznamy tak, aby mezi nimi nebyly mezery.

Počet záznamů v tabulce nebo kurzor zjišťuje funkce **`Rec_cnt`**. Vrácený počet zahrnuje i zrušené záznamy.

Stav záznamu v tabulce

Záznamy kterékoli tabulky mají systémový sloupec typu **Boolean** pojmenovaný `DELETED`. Hodnota tohoto sloupce říká, zda je záznam zrušený či nikoli. Pro zrušené záznamy má hodnotu `TRUE`, pro nezrušené (existující, platné) hodnotu `FALSE`.

Číst a přepisovat lze nejen platné, ale i zrušené záznamy. Do volných záznamů není dovoleno zapisovat a není definováno, co se z nich přečte.

Stav záznamu v kurzoru

Při otevírání kurzorů se do nich vybírají z tabulek pouze nezrušené záznamy. V editovatelném kurzoru se za dobu jeho existence může stát, že některé jeho záznamy budou zrušeny (buď majitelem kurzoru, nebo i jiným uživatelem v síti, není-li tabulka zamčena). Číslo záznamu v kurzoru je definováno pouze po dobu otevření tohoto kurzoru a pouze po tuto dobu je neměnné. Čísla stejných záznamů v kurzoru mohou být různá při každém otevření kurzoru.

Přiřazení mezi databázovým záznamem a proměnnou

Ve vnitřním programovacím jazyce lze přiřazovacím příkazem přenášet celý obsah záznamu mezi databází a proměnnou typu záznam **se stejnou strukturou**, jakou má databázový záznam. Databázový záznam může pocházet z tabulky, pevného nebo proměnného kurzoru.

Příklad:

```
TYPE curs_type = RECORD
      a : Integer;  s : string[12];
      ...
    END;
VAR
  cu : CURSOR curs_type;
  prom : curs_type;
  rec : trecnum;
BEGIN
   ...
   prom := cu[rec];
   ...
   cu[rec] := prom;
   ...
END.
```

Pokud by se mezi složkami typu `curs_type` vyskytl sloupec s proměnnou velikostí nebo multiatribut, do proměnné *prom* se nepřenese a prostřednictvím této proměnné s ním není možno pracovat.

Pozor!

Přestože databázové záznamy obsahují (jako první) sloupec `DELETED` typu **Boolean**, typ záznam používaný pro proměnné, jimž se přiřazují databázové záznamy, nesmí takovou složku obsahovat.

Sloupce

Ve vnitřním programovacím jazyce se sloupce označují svým jménem takto:

- sloupec tabulky je označen jménem uvedeným v definici tabulky;
- sloupec pevného kurzoru a proměnného kurzoru deklarovaného bez typu záznam se označuje jménem sloupce v odpovědi na dotaz (může být stejné jako jméno sloupce tabulky, automaticky přidělené jako např. EXPRn pro výraz nebo explicitně zadané pomocí klauzule AS);
- sloupec proměnného kurzoru deklarovaného s typem záznam se specifikuje pomocí jména příslušné složky typu záznam.

Přístup ke sloupci databázové tabulky můžete ve vnitřním programovacím jazyce použít na pravé i levé straně přiřazovacího příkazu. Je-li program znám typ sloupce, lze jej používat i ve výrazech.

Příklad

Máme tabulku firem FIRMY a tabulky faktur FAKTURY propojené přes sloupce ČÍSLO a FIRMA. Chceme projít nezaplacené faktury a zapsat do nich penále ve výši 5 procent částky. (Tato úloha zde slouží jako demonstrace k výkladu, pomocí příkazů SQL by se dala vyřešit jednodušeji.)

Řešení s pevným kurzorem

Je-li v aplikaci uložený dotaz Firmfak propojující firmy a faktury:

```
SELECT jméno, zaplaceno, částka, penále FROM Firmy, Faktury
WHERE číslo=firma
```

pak lze v programu využít pevný kurzor takto:

```
CURSOR Firmfak;
VAR r,cnt : trecnum;
begin
  if Open_cursor(Firmfak) then Signalize
  else begin
    Rec_cnt(Firmfak, cnt);
    for r:=0 to cnt-1 do if not Firmfak.zaplaceno[i] then
      Firmfak.penále[r]:=Firmfak[r].částka * 0.05;
    Close_cursor(Firmfak)
  end
end;
```

Řešení s proměnným kurzorem s typem

Není-li k dispozici vhodný dotaz, lze jej zadat až při otevírání kurzoru. Pokud popíšete strukturu kurzoru pomocí typu záznam, řešení bude vypadat takto:

```
VAR Ff : CURSOR record
  jm:string[50];  zaplac:Boolean;  částka,pokuta:Money;
END;
    r,cnt : trecnum;
begin
  if Open_sql_cursor(Ff,
     'SELECT jméno, zaplaceno, částka, penále'
     'FROM Firmy, Faktury WHERE číslo=firma') then Signalize
  else begin
    Rec_cnt(Ff, cnt);
    for r:=0 to cnt-1 do if not Ff.zaplac[i] then
      Ff.pokuta[r]:=Ff[r].částka * 0.05;
    Close_cursor(Ff)
  end
end;
```

Řešení s beztypovým proměnným kurzorem

Pokud nepopíšete strukturu kurzoru pomocí typu záznam, nelze provádět na jeho složkách žádné operace kromě přiřazení a je nutno využít pomocné proměnné.

```
VAR Ff : CURSOR;
    r,cnt : trecnum;
    z : Boolean;  m : Money;
begin
```

```
  if Open_sql_cursor(Ff,
     'SELECT jméno, zaplaceno, částka, penále'
     'FROM Firmy, Faktury WHERE číslo=firma') then Signalize
  else begin
    Rec_cnt(Ff, cnt);
    for r:=0 to cnt-1 do begin
      z:=Ff.zaplaceno[i];
      if not z then begin
        m:= Ff[r].částka;
        Ff.pokuta[r]:= m * 0.05;
      end
    end;
    Close_cursor(Ff)
  end
end;
```

Práce s délkou hodnoty sloupce proměnné velikosti

Na aktuální *délku* hodnoty sloupce odkážeme v programu tak, že za jméno sloupce připojíme symbol #.

Délka hodnoty sloupce proměnné velikosti je vždy typu **Integer**. Pokud tedy v tabulce (resp. kurzoru) TAB je sloupec POPIS typu **Text**, pak příkazy

```
délka := tab[por].popis#;
tab[por].popis# := 2480;
```

přečtou resp. změní délku hodnoty v tomto sloupci v záznamu s číslem POR.

Práce s hodnotou sloupce s proměnnou velikostí

Jediná operace, která je na hodnotách sloupců s proměnnou velikostí povolena, je přiřazovací příkaz, na jehož jedné straně stojí přístup ke sloupci a na druhé straně proměnná typu řetězec znaků nebo binární řetězec.

Se sloupcem s proměnnou velikostí se nepracuje jako s celkem, nýbrž se přenáší z něj nebo do něj určitý úsek. Tento úsek je popsán svým začátkem a svoji délkou takto: za přístup ke sloupci s proměnnou délkou umístíte speciální *dvojindex* ve tvaru:

```
[začátek, délka]
```

kde *začátek* je výraz typu **Integer** vyjadřující číslo prvního bajtu z hodnoty sloupce, který se má přenášet (číslováno od nuly) a *délka* je výraz typu **Short** vyjadřující délku přenášeného úseku.

Při čtení z databáze se nepřenese více bajtů, než kolik jich zbývá do konce hodnoty sloupce. Za poslední přenesený bajt do řetězce znaků se přidá ukončovací znak. Při zápisu do databáze se nepřenese více znaků, než kolik činí aktuální délka řetězce. Omezovací znak řetězce znaků se do databáze nezapisuje.

Příklad:

Budiž TX sloupec tabulky nebo kurzoru TAB typu **Text**. Program čte po úsecích hodnotu sloupce s proměnnou velikostí z prvního záznamu do řetězce a zapisuje ji do textového souboru.

```
const MAXX = 3000;
var f : file;
    s : CSstring[MAXX];
    start : Integer;  délka, úsek : Short;
begin
  délka := Tab[0].tx#;
  Rewrite(f, 'C:\longstr.txt');
  start := 0;
  while délka>0 do begin
    úsek := délka < MAXX ? délka : MAXX;
    s :=  Tab[0].tx[start,úsek];    //dvojindex
    Write(f, s);
    délka := délka - úsek;
  end;
  Close(f);
end.
```

Práce s multiatributy

S celými multiatributy najednou se nedají provádět téměř žádné operace. Proto se za specifikací multiatributu uvádí, se kterou složkou multiatributu chcete zacházet. Za přístup k multiatributu připojíte index v hranatých závorkách. Složky multiatributu se indexují vždy od nuly.

Je-li v tabulce (resp. kurzoru) TAB multiatribut MPOLE, pak na jeho sedmou složku v záznamu číslo POR odkážeme zápisem:

```
TAB[POR].MPOLE[6]
```

Při odkazu na neexistující složku multiatributu (zadáte-li index větší nebo roven počtu složek) reakce závisí na okolnostech:

- čtete-li neexistující hodnotu složky, pak obdržíte hodnotu NONE příslušného typu a server vydá varování, které můžete zjistit funkcí **Sz_warning**
- zapisujete-li do neexistující složky, pak je složka vytvořena nebo, pokud definice tabulky vytvoření této složky nepřipouští, dojde k běhové chybě.

Na počet složek multiatributu odkážete tak, že za jméno multiatributu připojíme místo indexu symbol #.

Tím získáme proměnnou typu **Short**, která obsahuje aktuální počet složek multiatributu. Zápisem do této proměnné lze počet složek zmenšit, nikoli však zvětšit. Nové složky multiatributu vznikají pouze zápisem nových hodnot. Například příkaz:

```
TAB[POR].MPOLE# := 0;
```

zruší všechny složky multiatributu.

Měnit počet složek multiatributu nelze v proměnném kurzoru, jehož struktura není v programu popsána. Při překladu programu pak totiž nelze zjistit, zda znak # označuje počet složek multiatributu nebo délku hodnoty sloupce proměnné velikosti.

Přechody přes ukazatel

Odkaz na záznam, na nějž odkazuje databázový ukazatel, lze získat tak, že za přístup k tomuto sloupci připojíme symbol ^. Za ním lze bezprostředně psát jméno sloupce v záznamu, kam ukazatel ukazuje.

Předpokládejme, že v tabulce (resp. kurzoru) TAB jsou sloupec UK a multiatribut MUK typu ukazatel do tabulky CÍL. V tabulce CÍL nechť existuje sloupec ATR. Pak zápisy:

```
TAB[POR].UK^.ATR
TAB[POR].MUK[I]^.ATR
```

odkazují na sloupec zpřístupněný ukazatelem.

Implicitní odkaz na sloupec

V některých případech se v programu vyskytují vícekrát poblíž sebe sloupce stejného záznamu určité tabulky resp. kurzoru. V takovém případě si programátor může ušetřit neustálé opakování jména tabulky a čísla záznamu pomocí databázové varianty příkazu **WITH**. Příkaz:

WITH *tabcur*[*por*] **DO** *vnitřní příkaz*

způsobí, že pokud se ve *vnitřním příkaze* vyskytne identifikátor sloupce tabulky resp. kurzoru *tabcur*, bude se považovat za přístup k sloupci záznamu s číslem *por*.

Je-li *tabcur* kurzor, musí být před provedením příkazu **WITH** otevřen.

Příkaz **WITH** se nedá se použít k práci se sloupci proměnného kurzoru, jehož struktura není popsána v jeho deklaraci. Použití příkazu **WITH** zrychluje běh programu pouze nepatrně.

Příklad:

V následujícím příkladu cyklus prochází všechny nezrušené záznamy v tabulce TAB (CNT a R jsou proměnné typu **Integer**):

```
Rec_cnt(tab, cnt);
FOR r:=0 TO cnt-1 DO
   WITH tab[r] DO
     IF NOT deleted THEN
     ..............
```

Při přístupu k standardnímu sloupci DELETED je zde využit příkaz **WITH**.

Specifika práce s kurzory

S kurzory lze v principu pracovat stejně jako s tabulkami. V závislosti na použitém dotazu však může kurzor ve srovnání s tabulkou umožňovat méně operaci:

- do kurzorů, které vznikají spojením dat z více tabulek, nelze vkládat záznamy ani nich rušit záznamy (případně je-li rušení povoleno, nepromítne se do databáze);
- data v kurzorech, která vznikají výpočtem (nejsou svázána s daty v některé tabulce) se nedají přepisovat.

Záznamy v kurzorech

Číslo záznamu v editovatelném kurzoru lze přepočíst na čísla záznamů v tabulkách, z nichž je kurzor vytvořen, pomocí funkce **`Translate`**. Díky ní lze mj. zjistit, zda záznam obsažený v kurzoru nebyl mezitím (jiným klientem) ve své tabulce zrušen. Přepočty čísel záznamů mezi kurzory umožňuje funkce **`Super_recnum`**.

Stav záznamu v kurzoru

Záznamy v *needitovatelných* kurzorech jsou vytvářeny z nezrušených záznamů v tabulkách a za dobu existence kurzoru se nemění. Výjimkou jsou pouze kurzory obsahující klauzuli HAVING, v nichž mohou být po otevření některé záznamy zrušené. Stav záznamu lze pak zjistit přečtením hodnoty sloupce `__DELETED` (dvě podtržítka na začátku).

O záznamech v *editovatelných* kurzorech je jisté, že v okamžiku otevření kurzoru jsou platné a odkazují na platné záznamy v tabulce resp. entice platných záznamů v tabulkách.

Za dobu existence kurzoru může dojít ke dvěma druhům rušení:

- Záznam v kurzoru je zrušen: Toto zrušení může provést pouze ten klient, který kurzor otevřel. Zrušený záznam poznáte tak, že zavoláte-li na něj funkci **`Translate`**:

  ```
  Translate(číslo_otevřeného_kurzoru,
            číslo_záznamu_v_kurzoru, 0, rec)
  ```

  pak v parametru *rec* obdržíte hodnotu -1.
- V tabulce byl zrušen záznam, na nějž odkazuje záznam v kurzoru: Takové zrušení může provést kterýkoli uživatel, pokud záznam není zamčen. Tuto situaci rozpoznáte tak, že zjistíte číslo záznamu v tabulce (z čísla záznamu v kurzoru pomocí funkce **`Translate`**) a pak zjistíte stav tohoto záznamu přečtením hodnoty jeho systémového sloupce `DELETED`.

Proti zrušení nebo přepsání lze záznam chránit jeho zamčením. Viz sekce o zamykání.

Postup vytváření kurzorů

Vytvářením kurzoru zde rozumíme přidávání záznamů do kurzoru. Kurzory mohou být vytvořeny celé najednou v okamžiku otevření, nebo mohou být vytvářeny postupně tak, jak program nebo uživatel požaduje jejich další a další záznamy.

Najednou budou vytvořeny *needitovatelné kurzory* a kurzory, které vyžadují *setřídění* záznamů ex post. Postupně se vytvářejí ty kurzory, v nichž lze záznamy resp. entice záznamů postupně vybírat z tabulky resp. entice tabulek a výsledek není nutno třídit - buď proto, že setřídění není specifikováno, nebo proto, že se záznamy z tabulky procházejí pomocí indexu v tom pořadí, v jakém mají být v kurzoru.

Postupné vytváření kurzoru je dokončeno nejpozději tehdy, když je třeba dojít na poslední záznam, např. při zjišťování počtu záznamů v kurzoru nebo při skoku na poslední záznamu ve formuláři do kurzoru.

Důležité:

Jakmile je vytváření kurzoru dokončeno, je množina záznamů obsažených v kurzoru definitivně stanovena a neovlivní ji přidávání nebo rušení záznamů v tabulkách, z nichž se záznamy vybíraly!

Množinu záznamů obsažených v kurzoru lze pak změnit pouze explicitním zrušením záznamu v *kurzoru* (nikoli v tabulce) nebo přidáním záznamu *do kurzoru* (nikoli do tabulky). Obě tyto operace vybíhají za standardní způsob práce z kurzorem.

Během práce s formulářem vedoucím do kurzoru lze provést akci, která způsobí zavření a nové vytvoření kurzoru a tím aktualizuje jeho obsah.

Subkurzory

Definicí kurzoru je určeno, jak má být z jedné nebo více tabulek obsažených v databázi vytvořena fiktivní tabulka. V mnoha situacích je třeba množinu záznamů obsažených v této fiktivní tabulce dále zúžit. Proto se při některých malých změnách v kurzoru vyplatí nevycházet od tabulek, ale od jiného kurzoru. Tato idea vedla k vytvoření konceptu subkurzorů.

Subkurzor je kurzor, který vzniká vybráním určitých záznamů z jiného, již otevřeného kurzoru.

Subkurzor tedy nevzniká podle definice jako normální kurzory. K otevření subkurzoru jsou potřebné:

- již otevřený kurzor (tzv. SUPERKURZOR) a
- dodatečná podmínka specifikující, které záznamy ze superkurzoru mají být vybrány do subkurzoru.

Superkurzor se nesmí uzavřít dříve, dokud není uzavřen každý jeho subkurzor. Podmínkou, která definuje subkurzor, je výraz typu Boolean vyhovující syntaxi SQL a vnitřního programovacího jazyka. Textový zápis tohoto výrazu se předává serveru při otevírání subkurzoru.

Subkurzor ze superkurzoru v e vnitřním jazyce vytváří funkce **Restrict_cursor** a k superkurzoru se lze vrátit funkcí **Restore_cursor**.

Příklad:

```
if Open_cursor(Adr_reg) then Signalize
else begin
  podm := "Adresar.cislo="+spom;
  if Restrict_cursor(Adr_reg, podm) then Signalize
  else begin
    if Print_opt(0) then Print_view("*S_hist",Adr_reg,-1,-1);
    Restore_cursor(Adr_reg);
  end;
  Close_cursor(Adr_reg);
end;
```

Použití subkurzorů pomocí funkce **Restrict_cursor** se vyplatí spíše pro pevné kurzory, kde není jiná možnost, jak kurzor dodatečně zúžit. Pro proměnné kurzory je často (časově) výhodnější původní kurzor zavřít a otevřít znovu s rozšířenou podmínkou.

Vyhledání záznamu

Vyhledání záznamů, které splňují určitou podmínku, lze provést některou z těchto cest:

- Projít všechny záznamy a na každém z nich ověřit, zda je podmínka splněna či nikoli. Tato cesta je nejpomalejší, ale hodí se i pro ty podmínky, které nedokážete zapsat ve formě výrazu.
- Předat podmínku serveru a vytvořit tak kurzor nebo subkurzor obsahující záznamy, které podmínku splňují. To je univerzální a nejrychlejší cesta, její rychlost závisí na přítomnosti vhodných indexů.
- Vyhledat záznam pomocí funkce **Loo*k*_up**. Tato cesta je nejsnadnější, hodí se pouze v případě, že kladete podmínku na hodnotu jediného sloupce a hledáte jediný záznam, který ji splňuje.

Příklad:

Použití funkce Look_up pro vyhledání záznamu z číselníku

```
begin
  ...
  zrusitAkci := true; branch := 0;
  Open_view("*VSTUP",NO_REDIR,MODAL_VIEW,0,0,id_vs);
  if not zrusitAkci then begin
    u := branch;       // ve form. zadaná hodnota
    recBranch := Looky_up(Obory,'cislo_o',u);
                       // nalezení odpovídajícího záznamu
```

```
if recBranch <> -1 then begin
  bookCode := obory[recBranch].prefix_o;
                  // přečtení dat ze záznamu
```

Příprava a provádění příkazů SQL

Databázový klient může provádět příkazy jazyka SQL dvojím způsobem:

- ***přímo***, tedy tak, že pomocí funkce **(cd_)SQL_execute** nebo direktiv **#sql**, **#sqlbegin** a **#sqlend** předává serveru příkaz, který má být proveden;
- ***spřípravou***, tedy tak, že nejprve pomocí funkce **(cd_)SQL_prepare** předá serveru příkaz, pak jej pomocí funkce **(cd_)SQL_exec_prepared** jednou nebo vícekrát provede, a nakonec přípravu zruší pomocí funkce **(cd_)SQL_drop**.

Výhodou **přímého** způsobu je jednoduchost na straně klienta. Volání jediné funkce vyvolá kompletní zpracování příkazu SQL.

Výhodou provádění SQL příkazů s **přípravou** je zejména úspora času, pokud se stejný příkaz (případně s různými hodnotami parametrů) má provádět opakovaně. Při zavolání funkce **(cd_)SQL_prepare** server kompletně analyzuje příkaz SQL a sestaví optimalizovaný plán pro jeho provedení. Každé zavolání funkce **(cd_)SQL_exec_prepared** pak již jen vyvolá uskutečnění tohoto plánu. Funkce **(cd_)SQL_drop** zruší již nepotřebný plán provádění příkazu.

Klient může mít v jednom okamžiku řadu připravených příkazů a může je provádět v libovolném pořadí.

Pokud připravený příkaz obsahuje odkazy na globální proměnné projektu klienta, pak se při provádění příkazu uplatní hodnoty proměnných platné v okamžiku volání funkce **(cd_)SQL_exec_prepared**. Díky tomu může opakované provedení připraveného příkazu pracovat pokaždé s jinými hodnotami.

Voláním funkce **(cd_)Set_application** se zruší příprava všech příkazů a zneplatní jejich handle.

Proměnné vnitřního programovacího jazyka v příkazech SQL

V příkazech jazyka SQL a dotazech lze používat také globálních proměnných deklarovaných v programu ve vnitřním jazyce, tedy na straně klienta **WinBase602**. V SQL těmto proměnným říkáme hostitelské proměnné, protože nejsou umístěny na serveru, ale v aplikaci využívající služby serveru. Díky nim lze zapsat dotaz a příkazy, jejichž obsah se bude měnit s hodnotami proměnných klienta.

Před každou takovou proměnnou s identifikátorem *id* je však nutno v SQL uvést dvojtečku. Navíc je možno přidat označení, zda jde o přenos hodnoty od klienta na server, ze serveru klientovi nebo oběma směry. Toto označení zefektivní přenosy dat mezi klientem a serverem a tím zrychlí práci. Proměnná se tedy zapíše takto:

Proměnná s obousměrným přenosem hodnoty:

:id nebo **:<>id** nebo **:><id**

Proměnná, do níž se přenese hodnota přiřazená serverem:

:>id

Proměnná, jejíž hodnotu může server přečíst:

:<id

V SQL není možno používat proměnných deklarovaných jako lokální v procedurách a funkcích. Typ proměnných musí odpovídat kontextu, v němž je proměnná použita, proměnné strukturovaných typů (např. pole) používat nelze.

Příklady použití

Příkaz UPDATE na vybraném záznamu

```
OS_CISLO := 1294;
PRIDAT := 500;
SQL_execute('UPDATE Zamestnanci SET plat=plat+:<PRIDAT WHERE
os_cis:<OS_CISLO');
```

Čtení hodnoty sloupce PLAT z vybraného záznamu do proměnné PL:

```
OS_CISLO := 1294;
SQL_execute('SELECT plat INTO :>PL FROM Zamestnanci WHERE
os_cis:<OS_CISLO');
```

Vložení nového záznamu:

```
OS_CISLO := 1294;
PL := 12000;
JMENO := 'NOVAK';
SQL_execute(
'INSERT INTO Zamestnanci(os_cis,plat,jmeno)
VALUES(:PL,:OS_CISLO,:JMENO)');
```

Volání procedury:

```
OS_CISLO := 1294;
SQL_execute('CALL JMENO_SEFA(:<OS_CISLO,:>SEF)');
```

Volání funkce:

```
OS_CISLO := 1294;
```

```
SQL_execute('SET :>MAX_PLAT = MAXIM_PLAT(:<OS_CISLO)');
```

Přístupová práva

Otázka práv uživatelů, jejich skupin a rolí k datům a objektům je pojednána v samostatné kapitole. V řadě aplikací lze vystačit s importem nastavení práv pro role během importu aplikace a následným obsazením uživatelů a skupin do rolí správcem aplikace. Manipulaci s právy pak není nutno programovat.

Pokud aplikační program potřebuje zjišťovat nebo měnit práva subjektů, pak může k tomu využít funkci **GetSet_privils**. Při volání této funkce se specifikuje:

- subjekt práva (uživatel, skupina nebo role);
- tabulka obsahující záznam nebo objekt, k němuž se práva vztahují;
- číslo záznamu, nastavují-li se práva k určitému záznamu, nebo -1, nastavují-li se globální práva pro celou tabulku;
- údaj o nastavení práv pro jednotlivé druhy práv a sloupce.

Nastavení práv k objektům (např. k definicím tabulek, formulářů apod.) se děje prostřednictvím nastavení práv k záznamům v tabulkách objektů.

Práva k obsahu tabulek lze také přidělovat příkazem SQL **GRANT** a odebírat příkazem SQL **REVOKE** (viz popis jazyka SQL).

Zápis nastavení práv

Hodnota aktuálního nastavení práv se zapisuje do pole o délce 65 bajtů. Práva jsou vyjádřena jednotlivými bity tak, že hodnota 1 značí přítomnost práva a 0 absenci práva. První bajt má význam pouze při specifikování práv tabulky jako celku (číslo záznamu je -1). Obsahuje údaje týkající se všech záznamů a všech sloupců. V dalších 64 bajtech jsou specifikována práva pro jednotlivé sloupce. Lze je zadat jak na úrovni celé tabulky, tak i pro určitý záznam.

Globální práva

Jednotlivé bity prvního bajtu mají tento význam:

Konstanta	bit	Význam
RIGHT_INSERT	4	právo přidávat nové záznamy do tabulky
RIGHT_DEL	8	právo rušit záznamy v tabulce
RIGHT_GRANT	128	právo poskytovat (vlastní) práva
RIGHT_NEW_READ	16	poskytnout právo číst k novým záznamům (*)
RIGHT_NEW_WRITE	32	poskytnout právo přepisovat k novým záz. (*)
RIGHT_NEW_DEL	64	poskytnout právo zrušit záznam k novým záz. (*)

Práva označená (*) se týkají záznamů, které budou v budoucnu vloženy do tabulky. Pro každý vložený záznam bude skupině EVERYBODY na základě nastavení těchto bitů přiděleno právo číst všechny sloupce, přepisovat všechny sloupce resp. zrušit tento záznam. Má smysl je přidělovat pouze skupině EVERYBODY funkcí **GetSet_privils**.

Práva ke sloupcům

Bity dalších bajtů v zápisu práv vyjadřují právo číst resp. přepisovat jednotlivé záznamy. Právo číst sloupec s číslem A je vyjádřeno X-tým bitem v N-tém bajtu a právo přepisovat Y-tým bitem v N-tém bajtu, přičemž platí:

N = (A-1) div 4 + 1
X = 2 * ((A-1) mod 4) + 1
Y = X + 1

Sloupce jsou číslovány od 1, bajty jsou číslovány od 1, bity bajtu jsou číslovány od 1.

POZOR! Při zjišťování čísla sloupce nezapomeňte na případné (skryté) systémové sloupce tabulky, které jsou umístěny před normálními sloupci!

Pro kompatibilitu se staršími verzemi jsou zachovány také funkce **Set_data_rights**, **Get_data_rights**, **Set_object_rights** a **Get_object_rights**.

Transakce a jejich zajištění

Havarijní situace a konzistence databáze

Při zpracování série požadavků klienta na server může dojít k situacím, které znemožní její dokončení. Může se jednat například o *havárii klienta, havárii serveru* (výpadek napájení), **přerušení spojení** mezi klientem a serverem, konflikt požadavků klienta s požadavky jiných klientů.

Pokud ze série souvisejících požadavků je provedena pouze část, může v databázi vzniknout nekonzistence, tedy logická rozpornost údajů.

Příklad:

Předpokládejme, že klient žádá o převedení jisté částky z jednoho účtu na jiný. Odečtením a přičtením stejné částky na dva účty se zachovává konzistence dat. Pokud se však z těchto operací provede pouze jedna, pak je konzistence porušena. Co hůře, klient vůbec nemusí být schopen takovou situaci řešit - provedení pouze druhé poloviny převodu (tedy uvedení databáze zpět do konzistentního stavu) může být právě v zájmu udržování konzistence přísně zakázáno.

Nekonzistence však mohou vznikat i tehdy, pokud klient provádí jedinou změnu. Například přepis jedné hodnoty v záznamu může vyvolat změny v řadě indexů, v nichž je

tato hodnota použita. Pokud by tyto změny byly provedeny neúplně, indexy by se staly nepoužitelnými.

Princip transakcí

K zabránění vzniku nekonzistencí v databázi je zaveden systém TRANSAKCÍ**.**

Princip práce systému

Klient, který chce provést sérii aktualizací, nejprve voláním funkce **`Start_transaction`** oznámí serveru zahájení transakce a pak vyšle nějaký počet požadavků. Po odeslání posledního z nich zavolá funkci **`Commit`**, která transakci uzavře. Server provádí požadavky klienta, ale nepromítá je do stavu databáze, dokud není zavolána funkce **`Commit`**. Pak zapíše do databáze všechny změny najednou.

Pokud klient změní údaj v databázi a před uzavřením transakce tento údaj čte, přečte změněnou hodnotu. Pokud však ve stejném okamžiku čte stejný údaj jiný klient, přečte původní hodnotu, a nová hodnota bude pro něj viditelná až poté, co první klient svou transakci úspěšně uzavře.

Pokud klient zavolá místo funkce **`Commit`** funkci **`Roll_back`***k*, jsou všechny změny provedené uvnitř transakce odvolány. Databáze je pak ve stejném stavu jako před zahájením transakce.

Pokud v průběhu provádění transakce dojde k jakékoli chybě (kromě chyby č. 131 - OUT_OF_TABLE), pak transakce okamžitě skončí a změny se do databáze nepromítnou a server odstraní efekt již provedených operací (jako kdyby byla provedena funkce **`Roll_back`***k*). Proto je v průběhu transakce nutno sledovat hodnoty všech prováděných funkcí.

Pokud klient nevolá funkce **`Start_transaction`** ani **`Commit`**, pak server považuje každý požadavek za samostatnou transakci. Proto po provedení každého požadavku promítne všechny změny do databáze.

Pokud v průběhu provádění transakce nedojde k žádné chybě, jsou všechny požadavky zaručeně provedeny.

Transakce nemůže programátor použít na akce prováděné s obsahem databáze prostřednictvím prezentační vrstvy, např. vkládání dat ve formuláři. Na této úrovni řídí spojování požadavků do transakcí pouze prezentační vrstva.

Vliv transakcí na rychlost

Spojováním požadavků do transakcí lze dosti podstatně urychlit činnost databáze. Blíže viz podkapitola *Efektivita programu* **v** této kapitole.

Jištění transakcí

Z výše uvedeného popisu transakčního mechanismu je zřejmé, že jediným kritickým bodem je provádění funkce **`Commit`** na serveru. Pokud by v jejím průběhu došlo k havárii, mohla by v databázi opět vzniknout nekonzistence.

Tomu lze zabránit tím, že je server provozován v režimu s JIŠTĚNÍM TRANSAKCÍ**. Tento režim může zvolit pouze administrátor.**

Při jištění transakcí je zaručeno, že se každá transakce provede buď celá, nebo se neprovede vůbec, a to bez ohledu na to, v jakém okamžiku dojde k havárii serveru.

Jištění transakcí zvětšuje téměř na dvojnásobek počet diskových operací při zápisu do databáze. Na čtení vliv nemá.

Potřeba jištění klesá, pokud je server provozován na počítači s nepřerušitelným zdrojem napájení.

Pokud operační systém odkládá zápis změn na disk a ponechává je ve vyrovnávací paměti, je jištění bezpečné pouze pokud zároveň zapnete **Zápis změn na disk při uzavření transakce**.

Podrobnější informace jsou v manuálu správce databáze.

Izolace transakcí

Pomocí funkce **`Set_transaction_isolation_level`** lze nastavit úroveň izolace transakcí pro klienta. Význam jednotlivých úrovní je popsán v kapitole o jazyce SQL.

Spolupráce klientů a zamykání informací

Pokud se serverem pracuje současně více než jeden klient, pak může docházet ke střetům jejich požadavků na operace se stejnými daty. Tyto konflikty mohou buď pozdržet provedení požadavku, anebo způsobit chybu při provádění požadavků.

Klient se tomu může bránit buď tím, že na data promyšleně umísťuje tzv. ZÁMKY, anebo v SQL zvolí vhodnou úroveň *izolace transakcí*.

Druhy a funkce zámků

Ve WinBase602 **existují zámky dvou druhů**: ZÁMEK PRO ČTENÍ **a** ZÁMEK PRO PŘEPIS**. Jejich názvy jsou poněkud zavádějící (vznikly historicky), proto věnujte pozornost vysvětlení jejich funkce.**

Zámek pro čtení

Zámek pro čtení (angl. *read lock*) zabraňuje všem ostatním klientům v přepsání dat, která jsou jim zamčená. Zajišťuje tedy neměnnost zamčených dat.

Zámek pro přepis

Zámek pro přepis (angl. *write lock*) klientovi zajišťuje, že bude moci přepsat zamčená data. Znemožňuje ostatním klientům jak přepsat data tak i zabránit mu přepsat data.

Zamčení dat nebude úspěšné, pokud na stejných datech již jsou zámky jiných klientů, s nimiž by byl nový zámek v konfliktu. Pro umísťování zámků platí toto pravidlo:

Na stejných datech smí být buď libovolné množství zámků pro čtení a žádný zámek pro přepis, anebo jediný zámek pro přepis a žádný další zámek (pro čtení ani pro přepis) od jiného klienta.

Klient může přepsat data pouze tehdy, pokud na nich není žádný zámek (pro čtení ani pro přepis) od jiného klienta. Jinak při pokusu o přepis dojde k chybě.

Co lze zamknout

Zamknout lze buď záznam nebo celou tabulku. Zamčení celé tabulky má stejný efekt jako zamčení všech záznamů v tabulce a má také stejné podmínky provedení. Pro zamčení tabulky pro čtení proto nesmí být zamčen žádný její záznam pro přepis, pro zamčení tabulky pro přepis nesmí být žádný její záznam nijak zamčen žádným jiným klientem.

Použití zámků v programu

Žádný zámek nebrání uživateli číst z databáze.

Význam zámku pro čtení

Zámek pro čtení klient použije tehdy, pokud potřebuje zachovat neměnnost určitých dat po určitou dobu. Například pokud klient po částech čte a zpracovává text uložený v databázi, pak jej zamkne pro čtení, aby jej jiný klient ve stejné době nemohl editovat.

Kdy budete zamykat

Nejčastěji se zámky používají před provedením zápisu nebo série zápisů do databáze. Zámky mají zajistit, že zápis bude úspěšně proveden. Nutnost testovat a řešit případné chyby se takto přesouvá z fáze provádění databázových aktualizací do fáze zamykání.

Kdy data nelze zamknout

Data nelze zamknout pro přepis, pokud jsou zamčena jiným klientem, pokud jsou právě přepisována nebo pokud byla někým přepsána a ještě neskočila transakce, v níž k přepisu došlo.

Nedaří-li se zamknout ta data, která potřebujete přepsat, pak má smysl pokus o zamčení vícenásobně zopakovat - lze očekávat, že soupeřící klient je natolik korektní, že data za-

krátko odemkne. Při déle se opakujícím neúspěchu bývá vhodné informovat uživatele a nechat další rozhodnutí na něm.

Zamykat a odmykat lze i zrušené záznamy v tabulkách.

Je-li zamčena celá tabulka, pak do ní nelze vložit nový záznam. Pokus o vložení záznamu skončí chybou. Vložení záznamu do zamčené tabulky by nemělo valný smysl, protože do tohoto záznamu by nebylo možno nic zapsat.

Pozor:

Velké množství zámků na záznamech stejné tabulky poněkud zpomaluje práci serveru. Proto je vhodnější místo současného zamčení mnoha tisíců záznamů zamknout rovnou celou tabulku resp. kurzor obsahující tyto záznamy. Znemožníte tím sice ostatním uživatelům **WinBase602** pracovat s ostatními (nepřepisovanými) záznamy během provádění operace, ale operace bude provedena podstatně rychleji. To platí pro všechny verze **WinBase602**.

Zámky a kurzory

Zamykat záznamy lze nejen v tabulkách, ale i v *editovatelných kurzorech*

Připomeňme si, že *editovatelný kurzor* odkazuje na data uložena v tabulkách a veškeré změny, které s daty provedete, se projeví na datech v tabulkách. Pokud kurzor obsahuje záznamy z jediné tabulky, pak zamčení záznamu v kurzoru znamená zamčení příslušného záznamu v tabulce. Pokud kurzor obsahuje záznamy z více spojených tabulek, pak zamčení záznamu v kurzoru znamená zamčení všech příslušných záznamů v tabulkách, z nichž je záznam v kurzoru odvozen.

Zrušené záznamy

Zrušený záznam v kurzoru nelze zamknout ani odemknout. Zrušením záznamu v kurzoru totiž zaniká jeho vazba na záznamy v tabulkách. Doporučujeme zamčené záznamy v kurzoru před zrušením odemknout.

Zamčení celého kurzoru

Zamčení celého editovatelného kurzoru znamená zamčení všech tabulek, které jsou v kurzoru použity (na jejichž data odkazuje), tedy nejen záznamů vybraných do kurzoru. Zamčení celého kurzoru doporučujeme používat velmi obezřetně.

Funkce pro zamykání a odemykání

Zamykání a odemykání dat provádějí funkce:

- **`Read_lock_table`**, **`Read_lock_record`**,
- **`Write_lock_table`**, **`Write_lock_record`**,
- **`Read_unlock_table`**, **`Read_unlock_record`**,
- **`Write_unlock_table`**, **`Write_unlock_record`**.

Transakce a zámky

Jakmile klient přepíše nějaký údaj, pak je záznam, který tento údaj obsahuje, zamčen pro přepis systémovým zámkem až do dokončení transakce (bez ohledu na to, zda klient tento záznam předtím explicitně zamknul).

Údaj nemůže být proto přepsán nebo zamčen nikým jiným, dokud transakce neskončí. Z tohoto důvodu není korektní provádět dlouho trvající transakce na sdílených datech. Je hrubou chybou čekat uvnitř rozpracované transakce na reakci uživatele.

V transakcích všeobecně platí, že dojde-li při jejich provádění k chybě, pak je odstraněn efekt všech dosud provedených aktualizací. Při zamykání dochází k chybám zcela běžně. Proto je rozumné zamknout vše, co bude v transakci přepsáno, ještě **před vstupem do transakce**.

Čekání na zámek

Pokud klient chce přepsat data nebo umístit zámek a tuto operaci znemožňuje zámek jiného klienta, pak požadavek bude na serveru čekat na uvolnění tohoto zámku. Aplikace by měly být navrhovány s ohledem na tento fakt - žádný klient by neměl dlouhodobě blokovat jiné klienty, není-li to absolutně nezbytné (například při hluboké restrukturalizaci dat).

Pomocí funkce **`waiting`** lze specifikovat, jak dlouho má klient čekat na uvolnění zámku nezbytného pro provedení jeho operace. Pokud specifikovaný čas vyprší a zámek stále není volný, požadavek skončí chybou. Typicky se v interaktivních operacích zadává pouze velmi krátké čekání, protože uživatel může nezdařenou operaci zopakovat, zatímco v neinteraktivních programech se čeká dlouho, neboť program na neúspěch svého požadavku obvykle nemůže dost dobře reagovat.

Implicitní hodnota doby čekání je pro normální klienty serveru **WinBase602** 0.5 sekundy. Pro replikační vlákna je nastaveno neomezeně dlouhé čekání.

Zablokování klientů

Příklad:

Pokud uživatel X chce zamknout záznamy A a B a uživatel Y chce zamknout záznamy B a A, pak může nastat tento scénář: Nejprve uživatel X zamkne záznam A, pak uživatel Y zamkne záznam B, a při pokusu o další zamykání se oba dostanou do nekonečného čekání na sebe navzájem.

Takový stav se nazývá ZABLOKOVÁNÍM (*angl.* deadlock). Server **WinBase602** jej rozpozná a vyřeší jej tak, že požadavek jednoho z klientů skončí chybou.

Při výskytu této chyby lze požadavek po čase zopakovat. Lepší je se pokusit vyhnout vzniku těchto chyb tak, že před každou operací vyžadující více zámků uživatel:

1. Pokusí se umístit všechny zámky bez čekání.
2. Pokud uspěl, provede operaci a zámky odstraní

3. Pokud neuspěl s některým zámkem, odstraní všechny zámky a po čase zkusí operaci znovu.

Odstraňování zámků

Klient nesmí zapomenout odstranit všechny zámky, jakmile je nezbytně nepotřebuje. Svými zámky totiž blokuje ostatní klienty v síti.

Odstranění zámků při odhlášení

Při odhlášení se klienta server odstraní všechny jeho zámky. Totéž se stane, pokud *vzdálený* klient havaruje. K automatickému odstranění zámků obvykle nedojde, pokud havaruje klient komunikující s *lokálním* serverem. Takové zámky může odstranit správce databáze.

Chyby při provádění operací

Provádění každé operace skončí buď úspěšně, nebo chybou. Mezi chyb patří i přerušení probíhající operace klientem.

Kromě chyb může server při provádění operací ohlásit také varování. Varování signalizuje, že operace byla provedena za nezvyklých podmínek. Varování samo o sobě není chybou, operaci lze považovat za úspěšně provedenou.

Dojde-li při provádění operace k chybě, jsou odvolány všechny změny, které operace v databázi již provedla. Pokud je operace prováděna v transakci, je odvolána celá transakce.

Návratová hodnota

O chybě informuje hodnota funkce volající službu serveru. Funkce serveru vracející hodnotu typu **Boolean**, vracejí `FALSE` při úspěchu a `TRUE` při chybě. Funkce vracející číslo záznamu vracejí při chybě číslo -1. *Pozor*: U funkcí uživatelského rozhraní tato pravidla neplatí.

Výsledek funkce doporučujeme v aplikacích vždy testovat. Tak lze odhalit chybu v okamžiku jejího vzniku, zatímco sledování chyby od projevů k jejímu původu může být velmi pracné.

Číslo chyby

Chyby i varování jsou očíslovány. Číslo chyby resp. varování vypovídá o příčině. Výskyt a číslo chyby resp. varování při provádění poslední operace lze zjistit zavoláním funkce **Sz_error** resp. **Sz_warning**.

Existují dvě speciální hodnoty funkce **Sz_error**:

- 255 - `NOT_ANSWERED` říká, že provádění operace není dosud dokončeno (k tomu může dojít pouze v režimu souběžného zpracování);

- 0 - NO_ERROR znamená, že zpracování proběhlo bezchybně.

Funkce **Sz_warning** má tyto speciální hodnoty:

- speciální hodnota 128 - IS_ERROR říká, že při zpracování došlo k chybě a tudíž nemá smysl se ptát na varování;
- hodnota 0 - NO_WARNING říká, že operace proběhla bez chyby i bez varování.

Pozor !

Funkce **Sz_error** a **Sz_warning** mají definovanou hodnotu pouze **bezprostředně** po volání některé funkce databázového jádra a po provedení akce v programu ve vnitřním jazyce, která funkci jádra volá (např. přiřazení hodnoty do databázového sloupce). Nedají se použít ke zjišťování výsledku funkcí patřících do prezentační vrstvy **WinBase602**.

K informování uživatele o výsledku poslední databázové operace nebo při ladění aplikace doporučujeme použít funkci prezentační vrstvy **Signalize**.

Ve vnitřním programovacím jazyce lze funkcí **Err_mask** určit, zda se při výskytu chyby v databázové operaci, která není explicitně vyvolána jako procedura či funkce (např. při provádění přiřazovacího příkazu do databázového sloupce) má ukončit program.

Důvody neúspěchu operací

K neúspěchu může při provádění operací dojít z řady důvodů. Důvody specifické pro jednotlivé funkce jsou popsány přímo u nich. Na tomto místě uvedeme důvody, které se vyskytují obecně.

Provedení operace nemůže být úspěšné, pokud klient dosud nenavázal spojení se serverem nebo se nepřihlásil.

Nejběžnější příčinou neúspěchu jsou nesprávné hodnoty parametrů.

Může jít například o nesmyslnou hodnotu předanou jako číslo tabulky nebo sloupce, číslo neexistujícího záznamu v tabulce atd.

Dalším běžným důvodem je, že provedení operace je znemožněno nedostatkem práv uživatele.

Provedení operace může (v řídkých případech) bránit nedostatek určitých zdrojů v jádře. Typickým příkladem je nedostatek operační paměti.

Při práci s kurzory je běžnou příčinou chyby opomenutí otevřít kurzor. Naopak opomenutí uzavřít kurzor sice bezprostředně nevadí, ale opakuje-li se, povede k vyčerpání systémových prostředků.

Interní rysy databázových objektů

Znalost obsahu této sekce není nezbytná pro programování velké většiny aplikací ve **WinBase602**. Objekty typické aplikace vzniknou při jejím importu do **WinBase602** a dále s nimi není nutno manipulovat.

Systémové tabulky

Server **WinBase602** používá pět systémových tabulek pro služební účely:

- TABTAB - tabulka všech tabulek;
- OBJTAB - tabulka všech objektů, které nejsou v ostatních systémových tabulkách;
- USERTAB - tabulka uživatelů a skupin;
- SERVTAB - tabulka registrovaných serverů, s nimiž lze replikovat;
- REPLTAB - tabulka replikačních pravidel pro jednotlivé aplikace, tabulky a cílové servery;
- KEYTAB - tabulka veřejných klíčů.

V prvních čtyřech tabulkách jsou uloženy všechny objekty obsažené v databázi. Každý záznam odpovídá jednomu objektu, absolutní číslo záznamu je číslem tohoto objektu. Záznam o objektu obsahuje jméno objektu, kategorii, označení aplikace, do níž objekt patří, definici objektu, různé příznaky a další interní informace.

Pomocí funkce **`Find_object`** lze na základně jména objektu nalézt číslo záznamu o kterémkoli objektu v příslušné tabulce. Funkce **`Find_object_by_id`** hledá číslo uživatele nebo serveru na základě jeho 12-bajtové identifikace.

S obsahem systémových tabulek aplikační programy zpravidla přímo nepracují. Existuje však několik výjimek. Zrušením záznamu v tabulce OBJTAB zrušíte objekt. Zápisem do záznamu o objektu lze objekt přejmenovat nebo přepsat jeho definici. V ostatních případech však s objekty manipulujete pomocí specializovaných funkcí. Práce s obsahem systémových tabulek podléhá omezením vyplývajícím z práv k objektům.

Sloupce systémových tabulek

Potřebujete-li přímo pracovat se sloupci systémových tabulek, pak ve vnitřním programovacím jazyce tabulku deklarujte a na sloupce se odkazujte jejich jménem.

Některé sloupce systémových tabulek, které lze použít v programu:

Jméno	OBJ TAB	TAB TAB	USER TAB	Typ	Číslo sloupce	Význam
TAB_NAME	--	ano	--	CSIString délky 31	OBJ_NAME_ATR	jméno tabulky

OBJ_NAME	ano	--	--	CSIString délky 31	OBJ_NAME_ATR	jméno objektu
LOGNAME	--	--	ano	CSIString délky 31	OBJ_NAME_ATR	jméno uživatele nebo skupiny
CATEGORY	ano	ano	ano	Char	OBJ_CATEG_ATR	kategorie objektu
APL_UUID	ano	ano	--	Binary délky 12 znaků	APPL_ID_ATR	identifikace aplikace
USR_UUID	--	--	ano	Binary délky 12 znaků	APPL_ID_ATR	identifikace uživatele (skupiny)
DEFIN	ano	ano	--	Nospec (OLE)	OBJ_DEF_ATR	textová definice objektu
FLAGS	ano	ano	--	Short	OBJ_FLAGS_ATR	různé příznaky objektu

Funkce **`Insert_object`** vloží záznam do systémové tabulky OBJTAB nebo USERTAB a zapíše hodnoty do sloupců OBJ_NAME resp. LOGNAME, CATEGORY a APL_UUID resp. USR_UUID.

Příklady použití systémových tabulek

Příklad:

Zjistit jména uživatelů databáze

```
SELECT logname FROM Usertab
WHERE (Usertab.category=chr(CATEG_USER))
```

Příklad:

Vybrat objekty patřící aplikaci "APLIKACE1"

```
SELECT T2.obj_name,Ord(T2.category)
FROM Objtab T1, Objtab T2
WHERE T1.apl_uuid=T2.apl_uuid
  AND (T1.obj_name="APLIKACE1"
  AND Ord(T1.category)=CATEG_APPL)
```

a

```
SELECT Tabtab.tab_name
FROM Objtab, Tabtab
WHERE Objtab.APL_UUID=Tabtab.APL_UUID
  AND (Objtab.obj_name="APLIKACE1"
  AND  Ord(T1.category)=CATEG_APPL)
```

Kategorie objektů

Tyto konstanty označují jednotlivé kategorie objektů **WinBase602**. Jsou to konstanty (typu **Integer**):

CATEG_TABLE	tabulka	0
CATEG_VIEW	pohled (formulář, sestava,...)	2
CATEG_CURSOR	pevný dotaz	3

CATEG_MENU	menu	6
CATEG_PGMSRC	zdrojový program	4
CATEG_PGMEXE	přeložený program	5
CATEG_APPL	aplikace	7
CATEG_PICT	Obrázek	8
CATEG_ROLE	role v aplikaci	10
CATEG_CONNECTION	ODBC napojení	11
CATEG_RELATION	Relace	12
CATEG_DRAWING	Schéma	13
CATEG_GRAPH	Graf	14
CATEG_REPLREL	replikační vztah	15
CATEG_PROC	procedura uložená na serveru	16
CATEG_TRIGGER	Trigger	17
CATEG_WWW	WWW objekt (šablona, konektor, ...)	18
CATEG_SEQ	Sekvence	20
IS_LINK	příznak spojovacího objektu	128
CATEG_USER	uživatel	1
CATEG_GROUP	skupina uživatelů	9

Tyto konstanty se používají v programovacím jazyce například jako parametry funkcí **`Find_object`**, **`Insert_object`**, **`Set_object_rig`***h*`ts`, **`Get_object_rig`***h*`ts`.

Pozor!

Sloupec `CATEGORY` v systémových tabulkách je typu **Char**, proto při porovnávání nezapomeňte provést konverzi funkcemi **`Ord`** nebo **`C`***h*`r`.

Příklad:

```
SELECT logname
FROM Usertab
WHERE (Usertab.category=Chr(CATEG_USER))
```

Vznik a zánik objektů

Autor aplikace, který je současně i jejím uživatelem, vytváří objekty zpravidla ručně pomocí interaktivních nástrojů vývojového prostředí **WinBase602**. V aplikacích, které jsou přenášeny mezi instalacemi **WinBase602** (např. jsou prodávány a kupovány) vznikají objekty importem během importu aplikace.

Mimo to mohou být objekty v odůvodněných případech vytvářeny a rušeny programem pomocí níže popsaných funkcí.

Každý nově vytvářený objekt vznikne v té aplikaci, která je právě otevřená.

Vytváření a rušení tabulek v programu

Program může vytvořit tabulku provedením SQL příkazu **CREATE TABLE**. Chcete-li v programu zrušit tabulku, pak použijte SQL příkaz **DROP TABLE**.

Příklad:

Příklad ukazuje zrušení tabulky ZBYTEČNÁ:

```
IF Sql_execute('DROP TABLE Zbytečná') THEN
  Info_box('Chyba SQL','Tabulka nesmazána');
```

Indexy

Program může k existující tabulce přidávat indexy SQL příkazem **CREATE INDEX** nebo rušit indexy SQL příkazem **DROP INDEX**.

Po vytvoření nebo zrušení tabulky pomocí SQL příkazu je nutno volat funkci **Re-list_objects**. Tato funkce zajistí správné naplnění seznamu tabulek na řídicím panelu aplikace. Bez jejího zavolání by se nová tabulka v seznamu neobjevila resp. zrušená tabulka by nezmizela.

Správa uživatelů, skupin a rolí

Nového uživatele vytvoříte voláním funkce **Create_user**. Heslo lze uživateli změnit funkcí **Set_password**. Novou skupinu uživatelů vytvoříte voláním funkce **Create_group**. Novou roli vytvoříte funkcí **Insert_object s** parametrem CATEG_ROLE.

Zařazovat uživatele do skupin a obsazovat uživatele a skupiny do rolí lze pomocí funkce **GetSet_group_role**.

Vytvářet skupiny a zařazovat do nich uživatele může pouze uživatel přihlášený jako správce databáze. Obsazovat do rolí smí pouze správce aplikace.

Interaktivní manipulaci s uživateli dovolují funkce **Acreate_user**, **Amodify_user** a **Aset_password**.

Vytváření a rušení ostatních objektů

U ostatních objektů postupujte takto:

1. funkcí **Insert_object** vložíte záznam do tabulky objektů
2. přiřazením zapíšete do sloupce DEFIN definici objektu v textovém tvaru (výjimkou jsou obrázky, jejich definice je uložena v některém z grafických formátů, např. BMP, PCX, GIF aj).

Příklad:

Vytvoření formuláře POHL1 z textové definice uložené v souboru POPIS.TXT:

```
pozice:=0;
Reset(f,'POPIS.TXT');
IF NOT Insert_object('POHL1', CATEG_VIEW, objnum) THEN
  WHILE NOT Eof(f) DO BEGIN
```

```
    Read(f,str);
    delka := strlength(str);
    Objtab[objnum].DEFIN[pozice,delka]:= str;
    pozice := pozice+delka;
  END;
Close(f);
```

Příklad:

Načtení obrázku do aplikace

```
procedure CopyFile(tb:ttablenum; rec, atr, index, handle:integer;
recode:integer);
external "WBPREZEN.DLL" name "copy_from_file";

function Open(var filename:string; par : integer) : integer;
external "KERNEL32" name "_lopen";

table
  OBJTAB;
const
  def_atr = 6;   // WinBase 5.0

var
  hfile : integer;
  ss : string[200];
  objnum : short;
begin
  ss := "*.bmp";
  if select_file(0,ss) then begin
    hfile := Open(ss,0);
    if not Find_object("newpicture",CATEG_PICT,objnum) then
      if Delete(objtab,objnum) then Signalize;
    if Insert_object("newpict2",CATEG_PICT,objnum) then Signalize
    else begin
       CopyFile(OBJTAB,objnum,def_atr,-1,hfile,0);
       Relist_objects;
    end;
  end;
end.
```

Rušení objektů

Objekt lze zrušit tak, že zrušíte záznam příslušného čísla v tabulce objektů (funkce **Delete**).

Příznaky objektů

Pomocí funkce **C*h*ng_component_flag** lze pro zvolený objekt z tabulky tabulek nebo tabulky objektů nastavit příznak, zda se má exportovat během exportu aplikace. Pro tabulky lze navíc nastavit příznak, zda se z nich mají exportovat data.

Nastavení startovního objektu aplikace

Startovní objekt aplikace lze předepsat nebo změnit funkcí **`Set_appl_starter`**.

Práce s připojenými a spojovacími objekty

Aplikační program může převzít kontrolu nad spojovacími objekty, které umožňují aplikaci pracovat s objekty patřícími jiné aplikaci.

Vyhledání spojovacího objektu

Při hledání objektu podle jména a kategorie pomocí funkce **`Find_object`** platí, že zadáte-li jméno spojovacího objektu a kategorii hledaného objektu, pak funkce vrátí již číslo připojeného objektu. Během hledání se totiž automaticky přejde po spoji vedoucím od spojovacího objektu k připojenému objektu.

Jak získáte číslo spojovacího objektu

Pokud byste chtěli získat číslo spojovacího objektu, pak musíte jako druhý parametr funkce **`Find_object`** uvést kategorii připojeného objektu sjednocenou s konstantou `IS_LINK`, např.:

```
Find_object('SPOJ_POHL', CATEG_VIEW or IS_LINK, viewnum);
```

Vyhledání spojovacího objektu je potřebné zejména tehdy, pokud jej chcete zrušit nebo přejmenovat. V ostatních případech potřebujete zpravidla pracovat s připojeným objektem.

Vytvoření spojovacího objektu

Spojovací objekty vytvářejí funkce **`Create_lin`***k* a **`Create2_lin`***k*. Vytvořením spojovacího objektu vznikne záznam v systémové tabulce, který ve sloupci `CATEGORY` má číslo kategorie sjednocené s konstantou `IS_LINK`. Ve sloupci `DEFIN` má odkaz na připojený objekt.

Spojovací tabulky zrušíte SQL příkazem **`DROP TABLE`**, ostatní spojovací objekty zrušíte funkcí **`Delete`** použitou na jejich záznam ve spojovací tabulce.

Příklad:

```
IF Find_object('SPOJ_POHL', CATEG_VIEW or IS_LINK, viewnum)
  THEN Signalize
ELSE Delete(OBJ_TABLENUM, viewnum);
```

Spojovací objekty v tabulkách objektů

Zvláštní pozornost je třeba věnovat spojovacím objektům, pokud procházíte záznamy v tabulce TABTAB nebo OBJTAB a hledáte tabulky resp. objekty patřící určité aplikaci. Mezi nimi najdete i spojovací objekty.

S čísly spojovacích objektů nelze pracovat jako s čísly plnohodnotných objektů, např. číslo spojovací tabulky uvedené jako parametr funkcí `Delete` **nebo** `Free_deleted` **by vedlo k chybě.**

Správný postup spočívá v tom, že si ke jménu spojovacího objektu naleznete pomocí funkce **`Find_object`** číslo připojeného objektu a pak pracujete s ním.

Efektivita programu

Při programování databázových aplikací se otázky efektivity řeší poněkud jinak, než v programech, které manipulují pouze s daty uloženými ve své operační paměti. Práce s obsahem databáze totiž zabírá tak velké procento času aplikace, že lze víceméně pominout čas výpočtů.

Rozdíl mezi čtením a zápisem

Prvním faktem, který si programátor musí uvědomit, je rozdíl mezi čtením a zápisem. Čtení z databáze je podstatně rychlejší. Proto, pokud při práci aplikace vznikají mezivýsledky, je rychlejší nezapisovat je do databázových tabulek, ale pouze je podržet v proměnných v operační paměti (pokud se tam vejdou).

Efektivita a transakce

Veškeré aktualizace obsahu databáze probíhají v tzv. TRANSAKCÍCH**. Na konci každé transakce se všechny provedené změny zapisují na disk.**

Řízení velikosti transakcí

Programátor má možnost řídit velikost transakcí pomocí procedur **`Start_transaction`** a **`Commit`** nebo pomocí příkazů jazyka SQL. Pokud této možnosti nevyužije, pak každá elementární operace modifikující obsah databáze vyvolá nový zápis na disk.

Příklad:

Předpokládejme, že chcete aktualizovat obsah tabulky, která se skládá z 1000 záznamů. Každý záznam nechť má 3 sloupce typu **Short**. Délka záznamu je tedy 7 bajtů a do diskového bloku o 512 bajtech se vejde 73 záznamů. Tabulka tedy zabírá 14 bloků. Pokud nepoužijete řízení transakcí, pak se provede celkem 3000 zápisů na disk, pro každou změnu sloupce jeden. Pokud však vytvoříte transakce zahrnující aktualizace přesně 73 záznamů nebo násobku 73, pak se provede pouze 14 zápisů bloku. Operací zápisu se tedy

provede 214-krát méně. Protože zápis na disk zabere většinu času běhu aplikace, dojde k celkovému zrychlení běhu v podobném poměru[2]).

Rizika a chyby

V použití transakcí se však skrývá určité riziko. Pokud při provádění transakce dojde k chybě v některé operaci, je celá transakce odvolána. Tím je odstraněn i efekt operací, které sice proběhly bezchybně, ale ocitly se ve stejné transakci s chybou.

Kde může dojít v odladěném programu k nečekané chybě? Typickým příkladem je výskyt hodnoty NULL na neočekávaném místě nebo porušení integritního omezení definovaného v tabulce.

Efektivita a zámky

Provádíte-li (s přihlédnutím k předchozímu odstavci) velmi mnoho přepisů v jedné transakci, je efektivnější předem zamknout tabulku, v níž se přepis provádí, zámkem pro přepis.

Všechny hodnoty, přepisované v transakci, zůstávají zamčeny až do uzavření transakce. Takto se mohou nahromadit na jedné tabulce statisíce zámků pro jednotlivé záznamy, a práce je pak pomalejší. Je-li předem zamčena celá tabulka, zámky pro jednotlivé záznamy se nepřidávají.

Indexy a kurzory

Faktorem podstatně ovlivňujícím efektivitu je existence indexů k tabulkám.

To se výrazně projeví při otevírání kurzorů, které specifikují podmínky na hodnoty sloupců, uspořádání nebo spojování tabulek. I zde bude rozdíl v rychlosti mnohonásobný (práce s velkými tabulkami bez indexů je prakticky nemožná).

Doporučení:

Není dobrým zvykem vytvářet kurzory, které obsahují velmi mnoho záznamů, a ty pak procházet po jednom. Navrhujte aplikace spíše tak, aby vám každý dotaz poskytl právě ty záznamy, které potřebujete.

Hromadné změny a indexy

Provádíte-li v tabulce hromadnou změnu hodnot ve *velmi mnoha* záznamech (např. pomocí SQL příkazu **`UPDATE`**) nebo import či zrušení mnoha záznamů, existence indexů operaci zpomaluje. V takových případech lze před provedením operace pomocí funkce

---

[2]) Tento příklad nebere v úvahu, že záznamy se neukládají v diskových blocích, ale clusterech, které jsou zpravidla větší než 512 bajtů. Tím se však na výpočtu efektivity nic podstatného nemění.

Enable_index vyřadit indexy z činnosti a po dokončení operace stejnou funkcí nechat indexy opětovně vybudovat.

Zrychlení operace může i mnohonásobně vyvážit čas potřebný na nové vybudování indexů.

Ostatní funkce serveru

Detailní popis všech funkcí naleznete v elektronické encyklopedii.

Diagnostické funkce

Níže uvedené funkce umožňují sledovat stav klienta a serveru. Slouží zejména pro diagnostické účely:

(cd_)Available_memory - funkce informuje o množství volné paměti u klienta nebo na serveru. Klesá-li množství volné paměti při každém zopakování určité operace, je to neklamný znak vážné chyby.

(cd_)Owned_cursors - funkce sděluje, kolik kurzorů má klient otevřeno. Pokud při opakování operace počet kurzorů stále roste, je to neklamné znamení chyby v aplikaci - nezavírají se kurzory.

(cd_)Database_integrity - funkce provede hloubkový test vnitřní integrity databáze.

(cd_)GetSet_fil_size - zastaralá funkce, která zjišťuje velikost databázového souboru nebo ji zvětšuje.

(cd_)GetSet_fil_blocks - funkce zjišťuje velikost databázového souboru nebo ji zvětšuje.

(cd_)Get_info - funkce vrátí řadu informaci o stavu serveru.

(cd_)Who_am_I - funkce vrátí uživatelské jméno, pod nimž je klient přihlášen na server.

(cd_)Am_I_db_admin - funkce zjišťuje, zda přihlášený uživatel patří do skupiny správců databáze.

(cd_)WinBase602_version - funkce vrací číslo verze **WinBase602**.

(cd_)Enable_task_switch - funkce povoluje nebo zakazuje doručování zpráv během čekání klienta na odpověď vzdáleného serveru. Někteří klienti fungují chybně, doručují-li se zprávy.

(cd_)Log_write - výpis zprávy do logu serveru.

(cd_)Message_to_clients - funkce rozešle zprávu všem síťovým klientům připojeným na stejný server.

(cd_)Connection_speed_test - funkce změří rychlost komunikace klienta se serverem.

(cd_)Appl_inst_count - funkce zjistí, kolik uživatelů má otevřenou stejnou aplikaci jako přihlášený uživatel

(cd_)Get_logged_user - funkce vrací informace o ostatních přihlášených uživatelích.

(cd_)Set_progress_report_modulus - funkce určuje, jak často se budou klientovi zasílat notifikace o průběhu SQL operace zpracovávající mnoho záznamů.

(cd_)Compact_database - funkce zmenši databázový soubor na nejmenší možnou velikost.

Jednou z nejběžnějších chyb, které způsobují nestabilní chování aplikací, je postupné vyčerpání systémových prostředků způsobené například neuzavíráním kurzorů. Pokud aplikace v některé opakující se operaci neuzavírá kurzory, dojde po určitém čase k vyčerpání systémové operační paměti.

Výše popsané funkce umožňují detekovat chybové stavy a tím odhalovat původ chyb.

Výpočet agregačních funkcí

Pro výpočet součtu, průměru, maxima, minima a počtu neprázdných hodnot určitého sloupce v kurzoru slouží funkce **(cd_)C_Sum**, **(cd_)C_Avg**, **(cd_)C_Max**, **(cd_)C_Min** a **(cd_)C_Count**.

Manipulace s tabulkou

Funkce **(cd_)Enable_index** povoluje resp. zakazuje použití indexů k tabulce. Když je použití indexů voláním této funkce povoleno, jsou indexy nově vybudovány, díky čemuž jejich obsah odpovídá aktuálnímu stavu tabulky. Při podezření na poškození indexů k tabulce je lze přebudovat tak, že se nejprve zakážou a pak povolí.

Funkce **(cd_)Compact_table** slouží ke zkompaktnění tabulky, tedy k „setřesení“ záznamů tak, aby mezi nimi nebylo žádné volné místo. Funkci nelze použít na tabulku, do níž vedou ukazatele.

Zastaralé funkce **Save_table** a **Restore_table** pro import a export dat jsou stále k dispozici, jsou však nahrazeny funkcí **(cd_)Move_data**.

Typové konverze

Aplikace může volat řadu funkcí pro konverzí typů a rozklad hodnot na složky. Externím jazykům jsou poskytnuty ty funkce, které pracují s typu specifickými pro **WinBase602**. Jde o funkce **Make_date**, **Day**, **Month**, **Year**, **Quarter**, **Day_of_week**, **Today**, **Make_time**, **Hours**, **Minutes**, **Seconds**, **Sec1000**, **Now**, **timestamp2date**, **timestamp2time**, **datetime2timestamp**, **Upcase**, **Money2real**, **Real2money**.

Funkce **WinBase602** pro konverze mezi řetězcem znaků a hodnotami různých typů se dají volat z externích jazyků, lze však také využít funkce zabudované v těchto jazycích.