Jazyk SQL ve WinBase602

Součástí **WinBase602** je varianta jazyka **SQL** (*Structured Query Language*). Tento jazyk je nejrozšířenějším standardem v oblasti definování schémat databází, zadávání operací nad daty a formulování dotazů. Jazyk sice není zcela vhodný pro běžného uživatele databáze, jeho síla však výborně slouží autorům databázových aplikací.

Tato kapitola popisuje jednotlivé oblasti SQL a jeho implementace ve **WinBase602**. Referenční popis příkazů a zabudovaných funkcí jazyka je v elektronické nápovědě.

Normy SQL

Implementovaná verze jazyka SQL vychází z normy SQL 2 (SQL 92), Intermediate level a z návrhu normy SQL 3 / PSM. Případné odlišnosti od této úrovně nebo od normy jsou vždy v textu explicitně uvedeny.

Pojmy

Pojmu SCHÉMA odpovídá ve **WinBase602** pojem APLIKACE. Hodnota NULL na serveru je v některých kontextech na straně klienta označována konstantami NONE... . Hodnota predikátu UNKNOWN je (v souladu s SQL 3) zaměnitelná s Booleovskou hodnotou NULL.

Kompatibilita s normami SQL a staršími verzemi WinBase602

Starší verze serveru **WinBase602** se v řadě případů chovaly odlišně, než předepisují nové normy jazyka SQL. Kupříkladu pokud je při vytváření odpovědi na dotaz požadována typová konverze, kterou nelze provést, starší **WinBase602** vracela hodnotu NULL, zatímco dle normy má celá operace skončit neúspěchem.

Aby byla zachována nezměněná funkce starších aplikací a přitom aby bylo možno vytvářet nové aplikace v souladu s normami SQL, umožňuje server **WinBase602** nastavit, do jaké míry se má zachovat kompatibilita funkce se staršími verze proti striktnímu dodržení norem. Toto nastavení lze libovolně měnit i za běhu jedné aplikace a pro různé klienty mohou ve stejném okamžiku platit různá nastavení. Změnu nastavení provádí klient voláním API funkce **`(cd_)Set_sql_option`**.

Popis syntaxe jazyka SQL v elektronické nápovědě

Pro zápis syntaxe používáme upravené BNF (Backus Naur Form). Syntaxe je popsána soustavou pravidel, na jejich levé straně stojí neterminál, jehož význam pravidlo definuje pomocí pravé strany. Mezi levou a pravou stranou pravidla je znak **`::=`**.

V pravidlech se neterminály zapisují *kurzívou*, terminály normálním písmem.

V pravidlech se vyskytují tyto metasymboly (symboly, které nejsou součástí syntaxe, ale umožňují zápis syntaxe):

| | oddělovač variant

[ ] závorky vymezující nepovinnou část

{ } metazávorky (ohraničují určitou část pravidla)

... symbol označující nula nebo více opakování bezprostředně předcházejícího symbolu (nebo více symbolů v metazávorkách)

Příklad použití zápisu:

```
příkaz_INSERT ::= INSERT INTO jméno_tabulky
    [(jméno_sloupce {,jméno_sloupce}… )] obsah
obsah ::= VALUES ( hodnota {, hodnota }… ) | DEFAULT VALUES |
    dotazový_výraz
```

Jak číst takovýto zápis

Příkaz `INSERT` (neboli neterminál *příkaz_INSERT*) je definován takto: začíná povinnými terminály (vyhrazenými slovy) `INSERT INTO` následovanými neterminálem *jméno_tabulky* (obsah neterminálů použitých na pravé straně definice je definován na následujících řádcích nebo v některém předcházejícím zápisu syntaxe nebo je jeho význam zřejmý jako v tomto případě).

Následuje nepovinná část uzavřená do hranatých závorek obsahující jeden nebo více neterminálů *jméno_sloupce*. Kulaté závorky nepatří mezi metasymboly, jsou tedy součástí příkazu. Složené závorky vymezují část, ke které se vztahuje symbol pro opakování.

Následuje neterminál *obsah* definovaný na následujícím řádku. Pomocí oddělovačů variant jsou naznačeny tři různé, navzájem se vylučující možnosti syntaxe. První způsob začíná vyhrazeným slovem `VALUES` a pokračuje výčtem jedné nebo více hodnot v kulatých závorkách, druhý způsob je definován pouze terminály `DEFAULT VALUES`, třetí způsob odkazuje na neterminál *dotazový výraz*. *Dotazový výraz* je jedním z nejdůležitějších pojmů SQL a je na něj odkazováno z mnoha míst, má proto svou vlastní stránku, na níž ukazuje odkaz níže v textu nápovědy.

Nedílnou součástí syntaxe je i popis a konkrétní příklad použití.

V příkladech dodržujeme tuto konvenci: zapisujeme jména tabulek s prvním písmenem velkým, jména sloupců malými písmeny, terminály SQL velkými písmeny. Tato konvence není povinná.

Lexikální elementy SQL

Znaky, vyskytující se v příkazech jazyka SQL, se interpretují dle kódové stránky 1250. Nejnižší úroveň syntaxe tvoří tzv. *lexikální elementy*.

Lexikálními elementy jsou identifikátory, vyhrazená slova, *literály* a komentáře.

Literály slouží k zápisu pevně dané hodnoty v textu příkazu SQL. Rozlišujeme je podle typu:

```
literál ::= číslo | znakový_řetězec | binární_řetězec |
zápis_data_a_času
```

Identifikátory

Identifikátory začínají písmenem nebo podtržítkem a obsahují písmena, číslice a znak „_“ (podtržítko). Přípustná jsou i písmena národní abecedy obsahující diakritické znaky. Počet signifikantních znaků v identifikátorech je 18, proto identifikátory, které se liší až za 18. znakem budou považovány za stejné. V identifikátorech se nerozlišuje mezi velkými a malými písmeny.

Identifikátory se musí lišit od tzv. vyhrazených slov, což je skupina identifikátorů s pevně daným neměnným významem. Vyhrazená slova jsou uvedena v nápovědě. Řada dalších slov má v SQL předem daný význam, nejsou však vyhrazena, a tudíž je lze volně používat ve funkci identifikátorů.

Lze také zapsat identifikátor, který obsahuje jiné než výše uvedené povolené znaky nebo který se shoduje s vyhrazeným slovem. Takový identifikátor musí být uzavřen do vymezujících znaků, kterými ve **WinBase602** jsou obrácené apostrofy. Zápisy `TO JE ONO!` nebo `UPDATE` jsou tedy platnými identifikátory.

Odlišnosti od normy

- Na rozdíl od SQL 2 se identifikátory, které obsahují nealfanumerické znaky nebo které jsou shodné s některým vyhrazeným slovem, neuzavírají do uvozovek, nýbrž do obrácených apostrofů.

Čísla

Čísla se zapisují obvyklým způsobem. Pokud místo desetinné tečky uvedete znak $ a za nim nejvýše dvě číslice, typ takového čísla bude Money.

Zápis reálného čísla může začínat desetinnou tečkou, interpretuje se jako "nula celá ...".

Příklady celých čísel: 0, 123456789, -987654321, +456

Příklady reálných čísel: 123.456, -123.456, 123.456e78, 123.456e-78, 1e12, .555,

Příklady čísel typu Money: 12$70, -57$45, 123456789$00

Znakové a binární řetězce

Znakové řetězce se zapisují v apostrofech. **WinBase602** sice také připouští jejich zápis v uvozovkách, je to však v rozporu s normami SQL. Pokud znakový řetězec má obsahovat znak apostrof nebo uvozovky, napíše se zdvojeně.

Binární řetězce se zapisují jako znak X a v apostrofech uzavřený výčet jednotlivých bajtů, z nichž každý je zapsán dvěma hexadecimálními číslicemi. Alternativně je lze zapsat

jako znak B a v apostrofech uzavřený výčet jednotlivých bajtů, z nichž každý je zapsán osmi číslicemi 0 nebo 1 reprezentujícími bity. Např. zápis X'C105' označuje binární řetězec skládající se ze dvou bajtů s hodnotami 193 a 5. Jeho alternativní zápis je B'1100000100000101'.

Odchylky od Intermediate level směrem k Full level nebo k SQL 3

- Binární a hexadecimální řetězce patří do Full level.

Rozšíření jazyka SQL proti normě

- Znakové a bitové řetězce lze zapisovat nejen v apostrofech, ale i v uvozovkách. Tohoto způsobu zápisu však nedoporučujeme využívat, protože uvozovky mají v normě SQL jiný význam.

Zápis času a data

Hodnoty typů DATE, TIME a TIMESTAMP lze zapisovat až trojím způsobem: dle norem SQL, dle specifikace ODBC a dle konvencí **WinBase602**. Tabulka shrnuje jednotlivé způsoby zápisu:

	Datum	**Čas**	**Datum s časem**
norma SQL	DATE'1962-04-27'	TIME'12:34:56.789'	TIMESTAMP'1962-04-27 12:34:56'
specifikace ODBC	{d '1962-04-27'}	{t '12:34:56.789'}	{ts '1962-04-27 12:34:56'}
styl WinBase602	27.4.1962	12:34:56.789	27.4.1962 12:34:56

Zápis podle specifikace ODBC ve složených závorkách není chápán jako komentář. Desetinná část sekund v zápisu času se ignoruje.

Použijete-li „styl WinBase602", pak lze z data vypustit rok - doplní se běžný rok.

Odchylky od Intermediate level směrem k Entry level

- V zápisech data ani času nelze specifikovat časovou zónu

Příklad

Vložení data, času a časové značky třemi způsoby:

```
procedure InsDat();
BEGIN
 INSERT INTO DatCas(dat,cas,datcas) VALUES
   (27.2.1998,12:34:56,27.2.1998 12:34:56);
 INSERT INTO DatCas(dat,cas,datcas) VALUES
   (DATE'1998-02-27',TIME'12:34:56',
    TIMESTAMP'1998-02-27 12:34:56');
 INSERT INTO DatCas(dat,cas,datcas) VALUES
   ({d'1998-02-27'},{t'12:34:56.789'},
    {ts'1998-02-27 12:34:56.789'});
END
```

Komentáře

Komentáře lze zapisovat dvojím způsobem. Buď komentář začíná *úvodním dvojznakem* a končí na konci řádky, nebo začíná *otevírací komentářovou závorkou* a končí *uzavírací komentářovou závorkou*.

Úvodním dvojznakem je buď dvojitá pomlčka -- nebo dvojité lomítko //.

Dvojicemi otevírací a uzavírací *komentářové závorky* mohou být /* a */ nebo { a }.

Zápisy začínající znakem { mají navíc specifický význam definovaný ODBC. Proto doporučujeme se jim ve funkci komentářů vyhýbat.

Rozšíření jazyka SQL proti normě

- Použití znaků //, { a } pro vymezení komentáře je rozšíření specifické pro WinBase602.

Datové typy

Tabulka shrnuje hlavní datové typy, které jsou k dispozici v jazyce SQL.

Typ	***Popis***
CHAR(*n*), VARCHAR(*n*)	řetězec znaků délky *n*
BINARY(*n*), VARBINARY(*n*)	binární řetězec délky *n*
CLOB	dlouhý znakový objekt
BLOB	dlouhý binární objekt
BIT, BOOLEAN	booleovská hodnota
INTEGER, INT	celé číslo
SMALLINT	malé celé číslo
DECIMAL(*n,d*), DEC(*n,d*)	číslo s desetinnou částí
NUMERIC(*n,d*)	číslo s desetinnou částí
REAL, FLOAT(*n*), DOUBLE PRECISION	reálné číslo
DATE	datum
TIME	čas
TIMESTAMP	datum s časem

Korespondence těchto typů s interními typy **WinBase602**, syntaxe a implementací definované vlastnosti jsou popsány v elektronické nápovědě.

Funkce

V SQL lze používat funkce patřící do čtyř skupin:

- standardní funkce definované normou SQL;
- vybrané standardní funkce vnitřního programovacího jazyka **WinBase602**;
- uživatelem definované funkce zapsané v jazyce SQL uložené na serveru;
- uživatelem definované funkce umístěné v externích knihovnách DLL.

Funkce z první a druhé skupiny se částečně překrývají, neboť vnitřní funkce **WinBase602** byly navrženy v době, kdy jazyk SQL nebyl v potřebné míře normován.

Z prostředí jazyka SQL nelze volat uživatelem definované funkce zapsané ve vnitřním programovacím jazyce klienta **WinBase602**.

Standardní funkce vnitřního jazyka WinBase602 dostupné z SQL

Všechny funkce, které bylo možno volat v SQL ve starších verzích **WinBase602**, zůstaly zachovány, i přesto, že jsou v SQL nahrazeny jinými funkcemi definovanými normou SQL (například funkci `Strpos` lze nahradit funkce `POSITION`). Jedná se o:

- Konverzní funkce:

`Str2int`, `Str2money`, `Str2real`, `Str2date`, `Str2time`, `Int2str`, `Money2str`, `Real2str`, `Date2str`, `Time2str`, `Datetime2timestamp`, `Timestamp2date`, `Timestamp2time`, `Ord`, `Chr`

- Matematické funkce:

`Odd`, `Abs`, `Iabs`, `Sqrt`, `Round`, `Trunc`, `Sqr`, `Isqr`, `Sin`, `Cos`, `Arctan`, `Ln`, `Exp`

- Funkce pro práci s datem a časem:

`Make_date`, `Day`, `Month`, `Quarter`, `Year`, `Today`, `Day_of_week`, `Make_time`, `Hours`, `Minutes`, `Seconds`, `Sec1000`, `Now`

- Řetězcové funkce

`Pref`, `Substr`, `Strcat`, `Strinsert`, `Strdelete`, `Strcopy`, `Strpos`, `Strlength`, `Strtrim`

- Ostatní funkce:

`WinBase602_version`, `Waits_for_me`, `Who_am_I`, `Current_application`

Tyto funkce nemají žádné vedlejší efekty na hodnoty svých argumentů. Funkce **`Strinsert`** nebo **`Strdelete`** tedy vracejí výslednou hodnotu, ale nemění řetězec předaný jako jejich argument (na rozdíl od volání stejných funkcí ve vnitřním programovacím jazyce klienta **WinBase602**).

Na serveru je také použitelná procedura **`Free_deleted`** a procedura **`Log_write`** (jakožto procedury nutno volat pomocí `CALL` např. (`CALL Free_deleted("MY_TABLE")`)

Funkce definované normou SQL

Lze volat následující funkce:

- `POSITION` - hledá subřetězec v řetězci nebo textu;
- `CHAR_LENGTH`, `OCTET_LENGTH`, `BIT_LENGTH` - vracejí délku řetězce;
- `EXTRACT` - vrací složku data nebo času;
- `SUBSTRING` - vybírá podřetězec z řetězce nebo textu;
- `UPPER`, `LOWER` - provádějí konverzi znakového řetězce nebo textu na velká resp. malá písmena;
- `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`- vrací momentální datum a čas;
- `NULLIF` - porovnává hodnoty výrazů;
- `COALESCE` - vrací hodnotu prvního výrazu, který nemá hodnotu NULL;
- `CAST` - přetypovává argument;
- Agregační funkce- funkce `AVG`, `SUM`, `MIN`, `MAX` a `COUNT`, počítají hodnoty nad množinou záznamů.

Výrazy v SQL

Ve VÝRAZECH v jazyce SQL lze používat:

- literály, tedy zápisy konstantních hodnot;
- jména sloupců tabulek, v jejichž kontextu se výraz vyhodnocuje (mohou se prefixovat jménem tabulky nebo aliasem);
- volání funkcí uložených na serveru;
- lokální proměnné platné v místě vyhodnocování výrazu;

- hostitelské proměnné, tedy odkazy na proměnné klienta;
- dynamické parametry (pouze přes rozhraní ODBC).

Kromě běžných operací lze ve výrazech použít konstrukci zvanou *podmíněný výraz CASE*. Slouží pro výběr jedné z více hodnot.

Při přístupu k hodnotám v tabulkách lze za jménem sloupce použít znak # pro označení aktuální délky hodnoty proměnné velikosti, index pro přístup k hodnotě multiatributu nebo dvojindex ve tvaru [start, délka] pro výběr úseku z hodnoty proměnné velikosti.

Aritmetické výrazy

V aritmetických výrazech lze používat binární operátor + pro sčítání, - pro odečítání, * pro násobení, / pro dělení, DIV pro celočíselné dělení a MOD pro zbytek po celočíselném dělení. Lze také používat unární minus - pro obrácenou hodnotu.

Pro porovnání numerických hodnot slouží operátory >, <, =, <=, >= a <>.

Při vyhodnocování výrazů dochází k implicitním konverzím typů operandů v rámci numerických typů. Konverze probíhá vždy na typ s vyšším rozsahem, tedy postupně SMALLINT na INTEGER na DECIMAL na REAL.

Rozšíření jazyka SQL proti normě

- Operátory DIV a MOD jsou specifické pro WinBase602.

Výrazy nad znakovými a binárními řetězci

Znakové řetězce lze spojovat operátorem zřetězení ||. Délka řetězce není omezena. Pro porovnání hodnot znakových nebo binárních řetězců slouží operátory >, <, =, =<, => a <>.

Znakové řetězce různých typů lze v operacích mezi sebou libovolně kombinovat.

Odchylky od Intermediate level směrem k Entry level

- Operátor zřetězení nelze použít na binární řetězce.

Implementací definované vlastnosti SQL ve WinBase602

- Operace nad dvojicemi řetězců se provádějí tak, že je-li některý operand typu **CSString**, pak se pracuje jako s operandy typu **CSString**, jinak je-li některý operand typu **CSIString**, pak se pracuje jako s operandy typu **CSIString**, jinak se pracuje jako s operandy typu **String**. Toto pravidlo ovlivňuje typ výsledku zřetězení a způsob vyhodnocení predikátu `LIKE`.

Výrazy nad datem a časem

K hodnotám typu DATE, TIME a TIMESTAMP lze přičítat a odečítat celé číslo ve významu intervalu, lze také počítat rozdíl dvou hodnot těchto stejných typů a obdržet celé číslo.

Interval pro typ DATE vyjadřuje počet dnů, pro typ TIME počet tisícin sekundy a pro typ TIMESTAMP počet sekund. Proto například interval získaný jako rozdíl dvou hodnot typu TIME je nutno před přičtením k hodnotě typu TIMESTAMP vydělit tisícem.

Odlišnosti od normy

- Použití celočíselného typu místo typu zvláštního typu INTERVAL je specifické pro WinBase602.

Podmínky a predikáty

PODMÍNKY jsou výrazy, které vracejí booleovskou hodnotu, tedy Ano-Ne-Nevím neboli TRUE, FALSE a UNKNOWN. Elementární podmínkou je predikát, složitější podmínky se z jednodušších vytvářejí pomocí operátorů OR, AND, NOT a závorek. Tabulku pravdivostních hodnot pro tyto operátory naleznete v nápovědě.

Predikáty shrnuje tato tabulka:

Predikát	***Význam***
>, <, <=, >=, =, <>	porovnání skalárních hodnot
SOME, ANY, ALL a >, <, <=, >=, =, <>	zobecněné porovnání
IS TRUE, FALSE nebo UNKNOWN	porovnání booleovských hodnot
IS NULL	porovnání hodnoty s NULL
BETWEEN	náležení hodnoty do intervalu
IN	náležení hodnoty do množiny hodnot
EXISTS	test neprázdného subdotazu
UNIQUE	test unikátních hodnot v subdotazu
LIKE	porovnání řetězců

Zobecněné porovnání porovnává hodnotu výrazu s množinou hodnot získanou jako odpověď na skalární *dotazový výraz*(subdotaz).

Predikát IN říká, zda hodnota jednoho výrazu je mezi hodnotami uvedených v seznamu nebo mezi hodnotami v odpovědi na skalární *dotazový výraz*(subdotaz).

Predikát EXISTS říká, zda v odpovědi na *dotazový výraz*(subdotaz) je alespoň jeden záznam.

Predikát UNIQUE říká, zda všechny záznamy v odpovědi na *dotazový výraz*(subdotaz) mají unikátní hodnoty. Přitom bere v úvahu pouze řádky obsahující hodnotu různou od NULL.

Predikát LIKE porovnává dva řetězce znaků s využitím „žolíkových“ znaků.

SQL umožňuje, aby součást *podmínky* tvořil další *dotazový výraz*, nazývaný zde zjednodušeně **subdotaz**. Subdotaz musí být uzavřen v kulatých závorkách. Jako subdotaz může stát kompletní dotazový výraz včetně dalších vnořených subdotazů.

Dotazy

DOTAZY slouží k nalezení v databázi nebo vypočtení množiny řádků, které odpovídají zadaným kritériím. Tuto množinu nazýváme odpovědí na dotaz. Pokud každý řádek obsahuje jedinou hodnotu, hovoříme o *skalárním dotazu*.

Nejznámější a nejpoužívanější forma dotazu se zapisuje ve tvaru SELECT... FROM... WHERE... .Tento tvar dotazu je popsán v sekci *specifikace dotazu*. **WinBase602** v souladu s SQL 2 umožňuje vytvářet složitější dotazové konstrukce spojováním jednodušších dotazů pomocí operátorů UNION, INTERSECT a EXCEPT. Tyto konstrukce se jmenují dotazové výrazy.

Ve funkci tabulky lze v dotazu použít nejen tabulku nebo odpověď na dotaz uložený v databázi (VIEW), také libovolný dotaz uzavřený v kulatých závorkách nebo spojení dvou tabulek pomocí některého druhu operace JOIN. Tyto možnosti popisuje sekce *zobecněná tabulka*.

Dotaz není sám o sobě příkazem, dá se však použít v řadě příkazů, např. INSERT, DECLARE CURSOR, CREATE VIEW. Je třeba rozlišovat mezi dotazem a příkazem SELECT INTO, který má podobnou syntaxi, ale slouží k přečtení jednoho řádku odpovědi na dotaz do proměnných.

Zobecněná tabulka

```
zobecněná_tabulka ::= { identifikátor_tabulky [ specifikace_indexu ] |
identifikátor_uloženého_dotazu | ( dotazový_výraz ) | join } [
přejmenování ]
specifikace_indexu ::= INDEX jméno_indexu
```

```
join ::= zobecněná_tabulka druh_joinu zobecněná_tabulka [
specifikace_joinu ]
druh_joinu ::= CROSS JOIN | [ NATURAL ] [ INNER | { LEFT | RIGHT | FULL
} [ OUTER ] ] JOIN
specifikace_joinu ::= ON podmínka | USING ( název_sloupce { ,
název_sloupce }…)
přejmenování ::= [ AS ] jméno [( jméno_sloupce { , jméno_sloupce }…)]
```

Pojem *zobecněná tabulka* definuje, které objekty mohou v jazyce SQL plnit funkci tabulky. Kromě skutečné tabulky nebo dotazu uloženého v databázi (VIEW) to může být i *dotazový výraz* v závorkách nebo *join.*

Pojem *join* vyjadřuje vytvoření spojených záznamů ze dvou tabulek zřetězením hodnot sloupců z jednoho záznamu první tabulky a jednoho záznamu druhé tabulky. *Druh joinu* a *specifikace joinu* určují, který záznam s kterým se takto spojí:

- `CROSS JOIN` znamená spojení každého záznamu s každým.
- `NATURAL` znamená, že obě zobecněné tabulky musejí mít sloupec stejného jména. Spojí se ty dvojice záznamů, které mají v tomto sloupci stejnou hodnotu. *Specifikace joinu* pak nesmí být uvedena.
- *specifikace joinu* ve tvaru `ON` *podmínka* znamená, že se spojí dvojice záznamů, pro něž je splněna zadaná *podmínka*. Podrobnosti o jejím sestavení viz podmínka.
- *specifikace joinu* ve tvaru `USING ( název_sloupce { , název_sloupce }…)` znamená, že se spojí ty dvojice záznamů, ve kterých jsou ve všech uvedených sloupcích stejné hodnoty. Sloupce zadaných názvů musí existovat v obou tabulkách. Jde tedy o zobecnění varianty `NATURAL`.
- není-li uvedena specifikace joinu ani `NATURAL`, spojuje se každý záznam s každým.
- `LEFT [ OUTER ] JOIN` znamená, že každý záznam první tabulky se musí spojit s některým (alespoň jedním) záznamem druhé tabulky. Není-li v druhé tabulce nalezen žádný záznam vyhovující podmínkám spojení, pak se záznam z první tabulky spojí s hodnotami NULL.
- `RIGHT [ OUTER ] JOIN` znamená, že každý záznam druhé tabulky se musí spojit s některým (alespoň jedním) záznamem první tabulky. Není-li v první tabulce nalezen žádný záznam vyhovující podmínkám spojení, pak se záznam z druhé tabulky spojí s hodnotami NULL.
- `FULL [ OUTER ] JOIN` znamená, že každý záznam z každé tabulky se musí spojit s některým (alespoň jedním) záznamem protější tabulky. Není-li v protější tabulce nalezen žádný záznam vyhovující podmínkám spojení, pak se záznam spojí s hodnotami NULL.
- `INNER JOIN` nebo `JOIN` znamená spojení těch záznamů, které vyhoví *specifikaci joinu*.

Specifikace indexu u jména tabulky vnutí použití konkrétního indexu bez ohledu na volbu optimalizátoru.

Případné *přejmenování* na konci *zobecněné tabulky* dovoluje zadat *jméno*, jimž se bude možno na tuto tabulku dále odkazovat, a případně také *jména sloupců* této tabulky. Počet uvedených *jmen sloupců* musí být stejný jako počet sloupců v zobecněné tabulce.

Odchylky od Intermediate level směrem k Full level nebo k SQL 3

- Použití dotazového výrazu v závorkách ve funkci zobecněné tabulky patří do Full level.

Rozšíření jazyka SQL proti normě

- Použití specifikace indexu je rozšířením WinBase602.

Specifikace dotazu

```
specifikace_dotazu ::= SELECT [ DISTINCT ] výraz { , výraz }…
FROM zobecněná tabulka { , zobecněná tabulka }…
[ WHERE podmínka_where ]
[ GROUP BY výrazg { , výrazg }… ]
[ HAVING podmínka_having ]
```

Specifikace dotazu vybere nebo vytvoří množinu záznamů ze záznamů uvedených *zobecněných tabulek*.

Je-li uvedena více než jedna *zobecněná tabulka*, provede se mezi nimi operace `INNER JOIN`, tedy každý záznam se spojí s každým. Jinak se použijí všechny záznamy *zobecněné tabulky*.

Je-li uvedena *podmínka_where*, ze získaných záznamů se vyberou pouze ty, na nich je tato podmínka splněna.

Je-li uvedena klauzule `GROUP BY`, záznamy se rozčlení do skupin tak, že každá skupina obsahuje záznamy se stejnými hodnotami všech *výrazůg*.

Je-li uvedena klauzule `HAVING`, vyberou se pouze ty skupiny, na nichž je splněna *podmínka having*.

Odpověď na dotaz zadaný touto specifikací tvoří sloupce zadané hodnotami *výrazů* na získaných záznamech nebo skupinách záznamů. Aby bylo možno vyhodnotit *výrazy* nad skupinami, vyžaduje se, aby každý výraz byl buď totožný s některým *výrazemg* nebo aby obsahoval odkazy na sloupce pouze jako argumenty agregačních funkcí.

V *podmínce where* se nesmí vyskytovat agregační funkce (s výjimkou subdotazů). Tyto funkce je možno použít pouze ve *výrazech* v klauzuli `SELECT` a v *podmínce having*. Po-

kud *specifikace dotazu* neobsahuje GROUP BY, pak se tyto agregační funkce počítají vzhledem k celé odpovědi na dotaz, jinak se počítají zvlášť pro každou skupinu.

Žádný *výrazg* nesmí být multiatributem.

Podrobnosti o sestavení podmínek ve *specifikaci dotazu* viz podmínka

Dotazový výraz

```
dotazový_výraz ::= { dotazový_term | dotaz_výraz { UNION | EXCEPT } [
ALL ] [ corresponding ] dotazový_term } [ třídění ]
dotazový_term ::= specifikace_dotazu | dotazový_term INTERSECT [ ALL ]
[ corresponding ] specifikace_dotazu
corresponding ::= CORRESPONDING [ BY ( jméno_sloupce {, jméno_sloupce
}… )]
třídění ::= ORDER BY výraz [ ASC | DESC ] {, výraz [ASC | DESC ] }…
```

Dotazový výraz umožňuje povést nad výsledkem *specifikace dotazu* operace relační algebry UNION, INTERSECT a EXCEPT a výsledek setřídit.

Operace UNION znamená sjednocení množin záznamů, operace EXCEPT rozdíl, operace INTERSECT průnik. Pokud není uvedeno ALL, pak každý záznam bude ve výsledku nejvýše jednou. Pokud je uvedeno ALL, pak záznam, který se vyskytuje M-krát v prvním operandu a N-krát v druhém operandu, bude ve výsledku operace UNION (M+N) krát, ve výsledku operace EXCEPT MAX(M-N, 0) krát, ve výsledku operace INTERSECT MIN(M, N) krát.

Pomocí specifikace *corresponding* lze zadat množinu sloupců ve výsledku operace:

- Pokud CORRESPONDING není uvedeno, pak se očekává, že oba operandy mají stejný počet sloupců a typy odpovídajících si sloupců jsou stejné. Tytéž sloupce budou i ve výsledku.
- Pokud v *corresponding* není uvedeno BY, pak ve výsledku budou pouze ty sloupce, které se v obou operandech jmenují stejné. Očekává se, že jsou i stejného typu a že alespoň jedna taková dvojice sloupců existuje.
- Pokud v *corresponding* je uvedeno BY, pak ve výsledku budou sloupce, jejichž *jména* následují na BY. Očekává se, že všechny tyto sloupce existují v obou operandech a že jsou stejného typu.

Výsledek *dotazového výrazu* se třídí podle hodnot *výrazů*, přičemž DESC znamená sestupné třídění, ASC nebo nic vzestupné. Pokud některý *výraz* má na dvou záznamech stejnou hodnotu, o pořadí záznamů rozhodné další *výraz*. Hodnoty NULL jsou při třídění považovány za menší než všechny ostatní hodnoty. *Výrazy* obsažené v klauzuli ORDER BY smějí obsahovat pouze jména sloupců *dotazového výrazu*, nikoli jména sloupců z tabulek použitých v *dotazovém výrazu*. *Výraz* definující třídění nesmí být multiatributem.

Rozšíření jazyka SQL proti normě

- Klauzule `ORDER BY` na tomto místě je rozšířením WinBase602.

Odchylky od Intermediate level směrem k Full level nebo k SQL 3

- Operace `UNION`, `EXCEPT` a `INTERSECT` je možno použít i v subdotazech

Příkazy jazyka SQL

PŘÍKAZY jazyka SQL jsou rozděleny do řady kategorií.

Příkazy pro definici dat

Příkaz	*Funkce*
`CREATE TABLE`	vytvoří novou tabulku
`ALTER TABLE`	změní definici tabulky
`DROP TABLE`	zruší tabulku
`CREATE INDEX`	vytvoří index k tabulce
`DROP INDEX`	zruší index k tabulce
`CREATE VIEW`	vytvoří dotaz uložený na serveru
`DROP VIEW`	zruší dotaz uložený na serveru
`CREATE SCHEMA`	vytvoří novou aplikaci a otevře ji
`DROP SCHEMA`	zruší celou aplikaci

Další příkazy z této kategorie jsou popsány v části o procedurách uložených na serveru a triggerech.

Příkazy pro manipulaci s daty

Příkaz	*Funkce*
`DELETE`	zruší záznamy v tabulce
`INSERT`	vloží záznamy do tabulky

UPDATE	modifikuje záznamy v tabulce nebo kurzoru
SELECT	sestrojí odpověď na dotaz
SELECT... INTO	zapíše odpověď na dotaz do proměnných

Další příkazy z této kategorie jsou popsány v části o transakcích a o kurzorech.

Příkazy pro manipulaci s právy

Příkaz	*Funkce*
GRANT	přidělí uživateli práva k tabulce
REVOKE	odebere uživateli práva k tabulce

Řídicí příkazy

Jazyk SQL ve **WinBase602** obsahuje kromě konceptů z SQL 2 také konstrukce pocházející z SQL 3/PSM. Jde o deklarace lokálních objektů a řídicí příkazy, s jejichž pomocí lze kombinovat provádění tradičních příkazů jazyka SQL, tvořit rutinya triggery.

Řídicí příkazy dovolují přiřazovat proměnným hodnoty, volat rutiny, vyvolávat a ošetřovat výjimky a vytvářet strukturované programové konstrukce.

Odchylky od Intermediate level směrem k Full level nebo k SQL 3

- Všechny řídicí příkazy jsou implementovány dle úrovně SQL 3.

K dispozici jsou tyto řídicí příkazy:

Příkaz	*Funkce*
BEGIN .. END	blok nebo atomický blok; může obsahovat lokální deklarace
SET	vyčíslení hodnoty výrazu a přiřazení do proměnné
IF	podmíněný příkaz
CASE	podmíněný příkaz
LOOP	cyklus
WHILE	cyklus s podmínkou na začátku
REPEAT	cyklus s podmínkou na konci
LEAVE	opuštění cyklu nebo bloku

FOR	cyklus přes všechny záznamy v kurzoru
CALL	volání procedury
RETURN	určení hodnoty funkce

Příkazy SIGNAL a RESIGNAL, patřící také do této kategorie, jsou popsány v části o práci s výjimkami.

Deklarace lokálních objektů v bloku

Jazyk SQL umožňuje deklarovat řadu objektů, na něž se mohou odvolávat příkazy SQL. Těmito objekty jsou proměnné, rutiny, výjimky, handlery a kurzory. Všechny tyto objekty se deklarují jako lokální uvnitř složeného příkazua pak jsou použitelné pouze v tomto příkazu. Mimo to lze rutiny deklarovat i globálně a pak jsou k dispozici celé databázové aplikaci - rutiny uložené na serveru.

Odchylky od Intermediate level směrem k Full level nebo k SQL 3

- Veškeré deklarace jsou implementovány dle úrovně SQL 3.

Procedury a funkce uložené na serveru

PROCEDURY A FUNKCE ULOŽENÉ NA SERVERU (označované souhrnně jako rutiny) zjednodušují tvorbu databázových aplikací a urychlují jejich běh. Klientská aplikace využívající uložených rutin pouze volá hotové rutiny (přitom jim předává vstupní parametry a přebírá výstupní), místo aby posílala serveru zvláštní požadavek na provedení každého příkazu obsaženého v rutinách.

Proceduru lze provést tak, že se zavolá pomocí SQL příkazu CALL. Funkce se provede, vyskytne-li se její volání jako součást výrazu. Během provádění funkce musí být v jejím těle proveden příkaz RETURN. Pokud proveden není, nastane chybový *sqlstate*2F001.

Rutinu lze vytvořit, modifikovat a zrušit:

- buď interaktivně ve vývojovém prostředí **WinBase602** pomocí nástrojů na řídicím panelu aplikace (editace interním nebo externím textovým editorem),
- nebo pomocí příkazů jazyka SQL CREATE, ALTER a DROP.

V obou prostředích se rutina definuje pomocí jazyka pro psaní rutin a triggerů z SQL 3. Ve vývojovém prostředí **WinBase602** začíná text editované rutiny slovem PROCEDURE nebo FUNCTION, slovo CREATE nebo ALTER se neuvádí.

Změna definice rutiny se projeví po uzavření transakce obsahující tento příkaz. I po tomto okamžiku pracují se **starou** verzí rutiny:

- příkazy zaslané serveru ve stejném požadavku klienta jako příkaz ALTER routine;

- dříve *připravené* příkazy odvolávající se na tuto rutinu (včetně příkazů připravených jinými klienty);
- dříve *otevřené* kurzory (včetně kurzorů otevřených jinými klienty).

Zrušená rutina skutečně zanikne až poté, co je dokončen požadavek klienta na server obsahující tento příkaz pro zrušení a co jsou zrušeny všechny připravené příkazy odvolávající se na tuto rutinu (včetně příkazů připravených jinými klienty).

Rozšíření jazyka SQL proti normě

- Ve **WinBase602** lze funkci volat příkazem `CALL` stejně jako proceduru. Při tomto způsobu volání se hodnota funkce ignoruje a provedení příkazu `RETURN` se nevyžaduje - je-li proveden, nemá žádný efekt.

Syntaxe rutiny

```
popis_rutiny ::= { popis_procedury | popis_funkce };
popis_procedury ::= PROCEDURE jméno ([ formální_parametr {,
formální_parametr }… ]) ; tělo
popis_funkce ::= FUNCTION jméno ([ formální_parametr {,
formální_parametr }… ]) RETURNS typ; tělo
formální_parametr ::= [ mód_parametru ] [ jméno ] typ [ DEFAULT výraz ]
mód_parametru  ::= IN | OUT | INOUT
tělo ::= příkaz | EXTERNAL NAME  označení_externí_rutiny
```

Každá rutina mu svoje jméno, pomocí něhož se volá. Dále rutina obsahuje výčet formálních parametrů, jimž se při volání rutiny přiřadí skutečné parametry. Po zavolání rutiny se provede její *tělo*.

Mód parametru určuje způsob předávání hodnoty parametru na začátku a konci provádění rutiny. Pokud tělo neobsahuje specifikaci `EXTERNAL`, pak platí tato pravidla: Při zahájení provádění rutiny se hodnoty skutečných parametrů s módy `IN` (vstupní parametr) a `INOUT` (vstupně-výstupní parametry) zkopírují do formálních parametrů. Po skončení provádění rutiny se hodnoty formálních parametrů s módy `OUT` (výstupní parametry) a `INOUT` zkopírují (zpět) do skutečných parametrů. Pokud mód není uveden, pak se v případě funkce předpokládá `IN`, v případě procedury se odvodí ze zacházení s formálním parametrem v těle procedury takto: parametry, jejichž hodnota se čte, mají mód IN, parametry, do nichž se zapisuje, mód `OUT`, a parametry, jejichž hodnota se čte i přepisuje, mód `INOUT`.

Pokud při volání rutiny není některý parametr uveden, pak příslušný formální parametr bude mít hodnotu *výrazu* uvedeného za `DEFAULT`. Není-li takový výraz uveden, bude hodnota parametru NULL. Je-li mód takového parametru `OUT` nebo `INOUT`, nezapíše se při ukončení rutiny hodnota formálního parametru nikam.

Jméno parametru lze ve specifikaci *formálního parametru* vynechat. Na takový parametr se však nelze v těle odvolávat. Vynechání jména má proto smysl pouze u externích rutin.

Hodnota funkce nesmí být typu Podpis, Historie, Datumovka, ukazatele nebo multiatribut. Typ formálních parametrů rutin nesmí být multiatribut.

Rozšíření jazyka SQL proti normě

- Ve **WinBase602** funkce smějí mít parametry s *módem* `IN`, `OUT` nebo `INOUT`, zatímco SQL 3 povoluje pouze *mód* `IN`.

Volání externích rutin a předávání parametrů

Rutina, v jejímž těle je specifikováno `EXTERNAL`, je implementována v externí knihovně DLL. *Označení externí rutiny* je řetězec znaků ve tvaru `'jméno_funkce@jméno_knihovny'`. Ve jméně knihovny lze uvést cestu - pokud cesta uvedena není, hledá se podle pravidel operačního systému. Pokud knihovna není nalezena nebo pokud v knihovně není nalezena funkce uvedeného jména, nastane chybový *sqlstate*38001.

O externích rutinách se předpokládá, že používají volací konvenci _STDCALL. Musí být umístěny v 32-bitové knihovně DLL. Volání 16-bitových rutin nebo rutin s jinou volací konvencí není možné.

Formální parametry označené `OUT` nebo `INOUT` se jim předávají referencí (tedy předává se adresa hodnoty parametru), formální parametry označené `IN` nebo neoznačené se předávají hodnotou. Označení a typ formálního parametru v deklaraci rutiny musí souhlasit s typem a způsobem předávání parametru v modulu, v němž je rutina implementována. Detaily hledejte v elektronické nápovědě.

Triggery

TRIGGER je nástroj, který zajišťuje automatické provedení programu v jazyce SQL při vložení, zrušení nebo změně záznamu v určité tabulce. Trigger může například zajistit, že:

- před vložením nového záznamu do tabulky bude provedena kontrola jeho obsahu a doplněny hodnoty některých sloupců;
- po smazání záznamu se provedou následné akce;
- při změně hodnoty určitého sloupce v záznamu se porovná stará a nová hodnota a na základě jejich vztahu se provedou další akce.

Triggery zjednodušují tvorbu aplikací, protože přenášejí část práce databázové aplikace na server. Umožňují centralizované definování pravidel platných pro informační systém.

Existuje-li například v podnikovém informačním systému tabulka zaměstnanců, lze pomocí triggerů popsat, jaké všechny akce musí být provedeny při přijetí nebo propuštění zaměstnance, změně platu nebo přeřazení do jiného oddělení. Tyto akce se naprogramují na jednom místě, ale budou sloužit všem aplikacím, které manipulují s tabulkou zaměstnanců. Dodržení pravidel pro údržbu evidence zaměstnanců pak bude zajišťovat server automaticky a konzistence dat bude zajištěna bez ohledu na možné chyby v měnících se aplikacích.

Trigger je pojmenovaným objektem patřícím do databázové aplikace. Má tyto vlastnosti:

- je svázán s určitou tabulkou a reaguje na změny v této tabulce;
- reaguje na právě jednu z SQL akcí `INSERT`, `DELETE` nebo `UPDATE`, triggery reagující na `UPDATE` mohou navíc specifikovat množinu sloupců, jejichž současná změna (všech vyjmenovaných) je spouští; trigger se spouští také prováděním obdobných akcí při práci s formuláři;
- provádí se buď před (`BEFORE`) provedením výše specifikované akce, nebo po ní (`AFTER`); pro triggery spouštěné jinak než SQL příkazem (práce s formuláři) platí tato výjimka: trigger spuštěný při vložení záznamu (`INSERT`) může být pouze `AFTER` (může např. zapsat hodnoty do vloženého záznamu);
- trigger nelze vyvolat editací hodnot multiatributu;
- může obsahovat podmínku, která se vyhodnotí před provedením triggeru a pokud není splněna, trigger nebude spuštěn;
- specifikuje akci (zapsanou v jazyce pro tvorbu rutin- PSM), která bude automaticky provedena, jakmile jsou splněny podmínky pro spuštění triggeru.

Při provedení konkrétní akce mohou být splněny podmínky pro spuštění více než jednoho triggeru. V takovém případě jsou spuštěny po řadě všechny. V současné podobně návrhu normy SQL 3 není zahrnut způsob, jak by uživatel v textu triggerů mohl definovat pořadí jejich spouštění. Je pravděpodobné, že před definitivní redakcí normy bude tento způsob specifikován a poté bude přidán i do WinBase602.

Provádění triggerů může měnit obsah databáze a tím spouštět další triggery. Spouštění triggerů se může do sebe libovolně zanořovat.

Pokud příkaz, který spustil trigger, je odvolán (kvůli chybě nebo v důsledku provedení příkazu `ROLLBACK`), pak také všechny akce, které v databázi trigger provedl, budou odvolány. Je-li trigger spuštěn v transakci, pak se provádí jako součást této transakce. Chyba, která nastane při provádění triggeru, má stejné důsledky, jako chyba v příkazu, který trigger spustil.

Uvnitř triggerů se nesmí provádět transakční příkazy `COMMIT` ani `ROLLBACK`, v opačném případě nastane chybový *sqlstate*2D000.

Při provádění triggerů se nekontrolují práva. Aby díky tomu nemohlo dojít k neoprávněnému zápisu, může triggery vytvářet pouze správce databáze resp. správce aplikace.

INSERT a UPDATE triggery se nespouštějí při importu dat do tabulky. DELETE triggery se nespouštějí při smazání tabulky jako objektu (při mazání záznamů samozřejmě ano).

Vytváření a modifikování triggerů

Triggery se vytvářejí, mění a ruší buď interaktivně na řídicím panelu **WinBase602** anebo pomocí příkazů CREATE TRIGGER, ALTER TRIGGER a DROP TRIGGER.

Vytvoření nového triggeru usnadňuje průvodce, v němž lze zadat základní vlastnosti triggeru a získat od něj hotovou kostru triggeru.

Pro modifikaci existujícího triggeru slouží interní textový editor. Pro editaci lze také použít externí textový editor, který zvolíte v parametrech **WinBase602** v menu ***Nástroje***. Externí editor se volá pomocí kontextového popup menu v seznamu objektů na řídicím panelu. Nedoporučujeme používat takové externí editory, které by do textu mohly vložit vlastní formátovací znaky.

Seznam triggerů k tabulce

Pokud na řídicím panelu aplikace označíte některou tabulku, v pravé části panelu lze pomocí záložky **Triggery** zobrazit seznam všech triggerů vztahujících se k tabulce. Poklepáním na jméno triggeru lze na něj otevřít textový editor.

Syntaxe popisu triggeru

```
popis_triggeru ::= TRIGGER jméno_triggeru [ BEFORE | AFTER ]
  akce ON jméno_tabulky [ reference ] [ granularita ]
  [ WHEN ( podmínka ) ] příkaz
akce ::= [ INSERT|DELETE|UPDATE [ OF sloupec { , sloupec }… ] ]
reference ::= REFERENCING { OLD [ ROW ] [ AS ] stará_řádka |
   NEW [ ROW ] [ AS ] nová_řádka }…
granularita ::= FOR EACH { ROW | STATEMENT }
```

Jméno triggeru musí být unikátní v rámci aplikace. *Jméno tabulky* musí označovat tabulku existující v rámci stejné aplikace a *sloupce* musí být jména vybraných sloupců této tabulky. Specifikace INSERT, DELETE nebo UPDATE říká, při jaké operaci s obsahem tabulky bude trigger spuštěn. Jsou-li za UPDATE uvedena navíc jména sloupců, bude spuštěn pouze při operacích modifikujících tyto sloupce, není-li seznam sloupců uveden, trigger se spustí při změně v každém sloupci.

Identifikátory *stará řádka* a *nová řádka* umožňují odvolávat se v podmínce a v příkazu na obsah databázového záznamu před a po provedení akce, která trigger spustila. Zápis ve tvaru stará_řádka.sloupec resp. nová_řádka.sloupec označuje starou resp. novou hodnotu sloupce. Je.li uvedeno INSERT, nesmí být uvedena *stará řádka*, je-li uvedeno DELETE, nesmí být uvedena *nová řádka*, neboť tyto pojmy nemají v těchto případech smysl. Identifikátory *stará řádka* a *nová řádka* nesmí být stejné.

Specifikace FOR EACH ROW říká, že trigger bude spuštěn pro každý změněný řádek zvlášť. Specifikace FOR EACH STATEMENT říká, že bude spuštěn pouze jednou v rámci provádění celého SQL příkazu, při němž se mění více řádků najednou. Není-li uvedeno

jedno ani druhé, míní se `FOR EACH STATEMENT`. Specifikace nové řádky a staré řádky je povolena pouze v případě, že je uvedeno `FOR EACH ROW`. Transientní tabulky triggerů nejsou ve WinBase602 zatím implementovány, proto varianta triggerů `FOR EACH STATEMENT` není příliš užitečná. V této verzi je proto doporučováno použít `FOR EACH ROW` vždy.

Při spuštění triggeru se nejprve vyhodnotí *podmínka*, a je-li splněná, provede se *příkaz*. V rámci *podmínky* ani *příkazu* nesmí být proveden příkaz otevírající nebo uzavírající transakci.

`BEFORE` trigger při svém provádění vidí v tabulce původní verzi dat, zatímco `AFTER` trigger vidí již změněnou verzi dat. Ostatní klienti vidí změny (způsobené příkazem SQL nebo triggery) až po uzavření transakce.

Efektivita triggerů

Provedení akcí pomocí triggeru reagujícího na příkaz SQL je zpravidla efektivnější než explicitní volání stejných akcí klientem. Nicméně v efektivitě provádění triggerů existují rozdíly:

- Nejefektivnější jsou triggery `AFTER INSERT` a `BEFORE DELETE`. Jejich provedení neznamená pro server prakticky žádnou dodatečnou zátěž.
- Dosti efektivní jsou také triggery `AFTER DELETE` a `AFTER UPDATE`. Jejich provedení vyžaduje pouze vykopírování určitých řádků z databáze do transientních proměnných.
- Triggery `BEFORE INSERT` a `BEFORE UPDATE` poněkud snižují rychlost provádění operací, protože nutí server, aby kvůli vytvoření správných transientních proměnných prováděl operaci `INSERT` resp. `UPDATE` méně efektivním způsobem.

V řadě situací nehraje roli, zda se použije `BEFORE` nebo `AFTER` trigger. Pak lze využít výše uvedených řádek k zefektivnění práce serveru.

Triggery spouštěné editací dat ve formuláři

Triggery definované v aplikaci lze spouštět i při interaktivní práci s formuláři vedoucími do dat, na něž byly triggery navrženy. Při navrhování je třeba uvažovat následující odlišnosti od triggerů, spouštěných přímo SQL příkazem.

INSERT trigger

Vložení nového záznamu ve formuláři se chová odlišně od vložení záznamu pomocí SQL příkazu `INSERT`. Nejprve se vloží "fiktivní" záznam, který má všechny hodnoty NULL, po zapsání hodnot se teprve provede kontrola integrity apod. `INSERT` trigger může reagovat pouze `AFTER` a nemůže odkazovat na žádné hodnoty, protože nejsou dosud vloženy. Může ale vložit do nového záznamu nějakou (implicitní) hodnotu.

Po zapsání hodnot do nového záznamu se uplatní `UPDATE` trigger.

UPDATE trigger

Chováním `UPDATE` triggeru je obdobné jako v SQL variantě. Je nutné pamatovat, že formulář se po provedení triggeru automaticky nepřekreslí.

Sekvence

SEKVENCE je objektem sloužícím ke generování posloupnosti celočíselných hodnot, zejména unikátních klíčů.

Sekvence není svázána s žádnou konkrétní tabulkou, takže se dá použít k vytváření klíčů, které budou unikátní nejen v rámci tabulky, ale i v rámci skupiny tabulek.

Kompatibilita

Sekvence nejsou součástí standardu SQL 92 ani SQL 3. Ve **WinBase602** byly sekvence implementovány v podobě shodné s ***Oracle 8***.

Sekvence se dají vytvářet, modifikovat a rušit buď z vývojového prostředí **WinBase602** nebo pomocí SQL příkazů `CREATE SEQUENCE`, `ALTER SEQUENCE` a `DROP SEQUENCE`. Jsou označeny svým jménem.

Při vytváření nebo modifikování sekvence lze zadat tyto údaje:

- počáteční hodnotu, od niž se generuje sekvence čísel;
- krok sekvence, tedy rozdíl mezi následující a minulou hodnotou v sekvenci;
- maximum, které nelze při generování sekvence překročit;
- minimum, které nelze při generování sekvence překročit (týká se klesajících sekvencí);
- příznak, zda sekvence má po dosažení maxima resp. minima pokračovat cyklicky (nevhodné pro generování unikátních klíčů);
- počet hodnot ve vyrovnávací paměti.

Na hodnoty generované pomocí sekvence se v SQL příkazech odkazuje pomocí zápisů:

jméno_sekvence`.CURVAL` a *jméno_sekvence*`.NEXTVAL`

NEXTVAL označuje novou hodnotu v sekvenci, CURVAL označuje hodnotu naposled vrácenou pomocí NEXTVAL.

Použití vyrovnávací paměti zrychluje generování hodnot v sekvenci. Dojde-li k výpadku systému, hodnoty v cache budou ztraceny. Implicitní velikost cache je 20.

Transakce a body návratu

V jazyce SQL lze členění akcí do transakcí dosáhnout pomocí příkazů `START TRANSACTION`, `COMMIT` a `ROLLBACK`. Kromě transakcí však existuje ještě jeden mechanismus dovolující odvolat skupinu provedených příkazů a vrátit se do stavu před jejich provedením. Tímto mechanismem jsou body návratu (`SAVEPOINT`).

Pomocí příkazu `SAVEPOINT` lze uvnitř transakce vytvořit bod návratu. Po provedení dalších příkazů lze se pak příkazem `ROLLBACK TO SAVEPOINT` vrátit k tomu stavu databáze, který byl v okamžiku provedení příkazu `SAVEPOINT`. Tento mechanismus je velmi podobný tomu, co se označuje jako vnořené transakce.

Příkaz `ROLLBACK` odvolává změny provedené v databázi. Neumožňuje odvolat přiřazení hodnoty lokální proměnné ani efekt provedených externích rutin (například odeslání dopisu).

Odchylky od Intermediate level směrem k Full level nebo k SQL 3

- Transakční příkazy jsou implementovány dle úrovně SQL 3.

Izolace transakcí

Klient databázového serveru může nastavit stupeň izolace vlastních transakcí od transakcí prováděných jinými klienty. Izolace transakcí určuje, v jaké míře se mohou změny v databázi souběžně prováděné různými klienty navzájem ovlivňovat.

Stupně izolace transakcí jsou definovány svojí schopností bránit vzniku tří fenoménů, k nimž by při souběžné práci klientů A a B mohlo docházet:

Fenomén **DIRTY READ**: Klient A provede změnu dat a prozatím neukončí transakci. Klient B přečte tato změněná data. Poté klient A odvolá svou transakci. Klient B tedy přečetl data, která nikdy nebyla potvrzena.

Fenomén **NON-REPEATABLE READ**: Klient A přečte data a prozatím neukončí transakci. Klient B změní nebo zruší tato data a ukončí svou transakci. Klient A ve své transakci znovu čte stejná data a nenajde je.

Fenomén **PHANTOM**: Klient A položí dotaz, přečte odpověď na něj a prozatím neukončí transakci. Klient B vloží do databáze další řádky vyhovující podmínkám v dotazu klienta A a ukončí svou transakci. Klient A ve své transakci znovu položí stejný dotaz a obdrží jinou odpověď.

Jednotlivé stupně izolace transakcí jsou definovány takto:

Stupeň izolace / fenomén	DIRTY READ	NON-REPEATABLE READ	PHANTOM
READ UNCOMMITTED	může nastat	může nastat	může nastat
READ COMMITTED	nemůže nastat	může nastat	může nastat
REPEATABLE READ	nemůže nastat	nemůže nastat	může nastat
SERIALIZABLE	nemůže nastat	nemůže nastat	nemůže nastat

Stupeň izolace SERIALIZABLE zaručuje, že souběžné transakce budou mít stejný efekt, jako by byly provedeny po sobě. Zároveň však snižuje schopnost serveru provádět akce souběžně, tudíž může snížit propustnost serveru, pokud více klientů pracuje se stejnými daty.

Stupeň izolace transakcí lze nastavit při zahájení transakce příkazem START TRANSACTIONnebo před zahájením transakce příkazem SET TRANSACTION.

Interně SQL server **WinBase602** implementuje stupně izolace READ COMMITTED a SERIALIZABLE. Při nastavení READ UNCOMMITTED se vnitřně nastaví READ COMMITTED, při nastavení REPEATABLE READ se vnitřně nastaví SERIALIZABLE. Toto chování je plně v souladu s normou.

Příkazy pro řízení transakcí

Transakce se řídí těmito příkazy:

Příkaz	***Funkce***
START TRANSACTION	zahajuje novou transakci a určuje její druh a stupeň izolace
COMMIT	ukončuje právě probíhající transakci
ROLLBACK	ukončuje právě probíhající transakci
SET TRANSACTION	nastavuje druh a stupeň izolace následující transakce
SAVEPOINT	vytváří bod návratu
ROLLBACK TO SAVEPOINT	vrací databázi do stavu, v níž byla při vytvoření bodu návratu
RELEASE SAVEPOINT	ruší uvedený bod návratu

Kurzory v SQL

KURZORY v SQL jsou nástrojem, jak programově zpracovat odpověď na *dotaz*. Umožňují číst řádky této odpovědi, modifikovat je nebo rušit je. Kurzor je vždy nastaven na některé řádce odpovědi, je před některou řádkou nebo za poslední řádkou odpovědi.

Lokální kurzory lze deklarovat v každém složeném příkazu. Mimo to lze pracovat s globálními kurzory, které se dají otevřít do libovolného dotazu uloženého v databázové aplikaci.

Deklarace kurzoru

```
deklarace_kurzoru ::= DECLARE jméno [ INSENSITIVE | SENSITIVE ]
  [ SCROLL ] CURSOR FOR specifikace_kurzoru;
specifikace_kurzoru  ::= dotazový_výraz FOR { READ ONLY |
  UPDATE [ OF sloupec {, sloupec }… ] }
```

Je-li v deklaraci uvedeno INSENSITIVE, pak změny v datech provedené prostřednictvím kurzoru se v databázi neprojeví. Kurzor bude pracovat s vlastní tabulkou obsahující odpověď na *dotazový výraz*. Je-li uvedeno SENSITIVE, pak se změny v datech provedené prostřednictvím kurzoru projevují v databázi, což je implicitní stav. SENSITIVE nesmí být uvedeno v deklaraci kurzoru obsahujícího grupování.

Specifikace SCROLL požaduje přístup k datům v kurzoru v libovolném pořadí. Tuto vlastnost mají ve **WinBase602** všechny kurzory a proto se SCROLL ignoruje.

Klauzule FOR UPDATE se vylučuje s INSENSITIVE a dá se použít pouze na editovatelné kurzory. Klauzule FOR READ ONLY zabraňuje v editaci dat a vkládání nebo rušení záznamů pomocí tohoto kurzoru.

Odchylky od Intermediate level směrem k Full level nebo k SQL 3

- Specifikace INSENSITIVE patří do úrovně Full SQL.
- Klauzuli FOR UPDATE je možno libovolně kombinovat se SCROLL a ORDER BY.

Použití kurzorů

S kurzory pracují tyto příkazy SQL:

Příkaz	*Funkce*
OPEN	otevře kurzor a nastaví se před první řádek odpovědi
FETCH	nastaví kurzor na určený řádek a přečte hodnoty ze sloupců
UPDATE CURRENT OF	provede změny na běžné řádce v kurzoru

DELETE CURRENT OF	zruší běžnou řádku v kurzoru
CLOSE	uzavře kurzor

Každý kurzor musí být nejprve otevřen příkazem OPEN. Poté lze procházet a číst jeho řádky příkazem FETCH, odvolávat se na jeho běžný záznam v příkazech UPDATE CURRENT OF a DELETE CURRENT OF a nakonec jej uzavřít příkazem CLOSE.

Neuzavřené lokální kurzory jsou automaticky uzavřeny při opuštění složeného příkazu, v němž jsou deklarovány.

Jiný způsob, jak otevřít kurzor, projít a zpracovat postupně všechny jeho záznamy a kurzor uzavřít, nabízí řídicí příkaz FOR. Při použití FOR není třeba deklarovat NOT FOUND handler.

Detailní popis kurzorů v SQL naleznete v elektronické nápovědě.

Chyby, výjimky a jejich ošetřování

Chyby a sqlstate

Norma SQL předepisuje, že chybové stavy a varování mají být označovány pomocí tzv. *sqlstates*, což jsou řetězce pěti znaků složené z číslic a velkých písmen latinské abecedy. Norma pro řadu chyb předepisuje, který *sqlstate* má při jejich výskytu nastat. Mimo to ponechává prostor pro *sqlstates* definované implementací.

Ve **WinBase602** jsou chyby interně označovány číslem. Některé z těchto chyb odpovídají některému *sqlstate*, jiné jsou specifické po **WinBase602**. Vzhledem k tomu, že pro většinu syntaktických chyb a všechny chyby z nedostatku práv existuje jediný *sqlstate*, poskytují čísla chyb **WinBase602** detailnější rozlišení.

Na *sqlstates* se v příkazech SQL odvolávají deklarace výjimek a handlerů. Chcete-li zachytit a zpracovat určitou chybu, musíte znát její *sqlstate*. Tabulka v elektronické nápovědě specifikuje *sqlstates* odpovídající jeden k jedné chybám **WinBase602**. Ostatní (implementací definované) chyby se vyjadřují pomocí *sqlstate* ve tvaru „W*abcd*“, kde *abcd* je čtyřmístné číslo chyby, doplněné v případě potřeby zleva nulami. Například chybě 171(deadlock) odpovídá *sqlstate* W0171.

Výjimky

VÝJIMKAMI nazýváme nestandardní situace, které mohou vzniknout při provádění příkazů SQL. Příkladem budiž například provádění příkazu FETCH v situaci, kdy požadovaný záznam neexistuje. Taková situace obvykle znamená vznik chyby a její oznámení klien-

tovi. Jazyk SQL však disponuje nástroji, jak výjimky ošetřit, tedy předepsat, co se po vzniku výjimky má dít, a zabránit tak vzniku chyby.

Výjimky jsou trojího druhu:

1. Výjimky odpovídající chybovým stavům serveru *sqlstate* a označované značkou stavu, například "3C000". Tyto výjimky nastávají, dostane-li se server při plnění příkazů klienta do příslušného stavu.
2. Výjimky deklarované uživatelem v deklaraci `CONDITION` a označené identifikátorem. Tyto výjimky nastávají, provede-li se v programu příkaz `SIGNAL`, jsou tedy spouštěny explicitně. Slouží k opuštění posloupnosti příkazů, které se právě provádějí, opuštění jednoho nebo více složených příkazů nebo jedné či více rozpracovaných rutin.
3. Výjimky deklarované uživatelem v deklaraci `CONDITION`, označené identifikátorem a přiřazené určitému stavu serveru *sqlstate*. Tyto výjimky nastávají buď dostane-li se server při plnění příkazů klienta do příslušného stavu, anebo při explicitním provedení příkazu `SIGNAL`. Spojují v sobě vlastnosti prvního a druhého typu.

Odchylky od Intermediate level směrem k Full level nebo k SQL 3

- Výjimky a jejich ošetření jsou implementovány dle úrovně SQL 3.

Deklarace výjimky

```
deklarace_výjimky ::= DECLARE identifikátor CONDITION
    [ FOR SQLSTATE [ VALUE ] sqlstate ];
```

Touto deklarací se deklaruje výjimkaoznačená *identifikátorem*. Deklarovanou výjimku lze vyvolat příkazem `SIGNAL` a lze jí přiřadit handler v deklaraci handleru. Je-li uveden řetězec znaků *sqlstate*, pak se deklarovaná výjimka s ním sdruží. Případný handler deklarovaný pro tuto výjimku bude platit i pro uvedený *sqlstate*. Stejný *sqlstate* se nesmí použít ve více výjimkách deklarovaných se stejným rozsahem platnosti.

Handlery výjimek

Nastane-li výjimka, systém pro ní hledá tzv. HANDLER. Handler se přiřazuje výjimce nebo třídě výjimek v deklaraci handleru. Při hledání handleru se postupuje od místa, v němž k výjimce došlo, směrem ven. Je-li handler nalezen, provede se a v dalším provádění příkazů se pokračuje v závislosti na druhu handleru.

Dojde-li při provádění handleru H k výjimce, která není uvnitř něj ošetřena, neuplatní se při jejím ošetření handlery deklarované na stejné úrovni jako handler H, ale pouze handlery deklarované vně.

Uvnitř handleru lze použít příkaz `RESIGNAL`, který předá výjimku zpracovávanou tímto handlerem k vnějšímu ošetření.

Handlery přiřazené k ošetření výjimek mohou zpracovávat také chyby definované implementací **WinBase602**. Na tyto chyby se deklarace handleru odkazuje pomocí *sqlstate* ve tvaru W*abcd*, kde *abcd* je číslo chyby, případně doplněné zleva nulami.

Handlery nemohou ošetřovat chyby v syntaxi příkazů SQL.

Deklarace handleru

```
deklarace_handleru ::= DECLARE { CONTINUE | EXIT | REDO | UNDO }
    HANDLER FOR výjimka {, výjimka }…  příkaz;
výjimka ::= identifikátor_výjimky | SQLSTATE [ VALUE ] sqlstate
    | SQLEXCEPTION | SQLWARNING | NOT FOUND
```

Touto deklarací se handler zadaný *příkazem* přiřadí ke specifikovaným výjimkámnebo třídám výjimek a může se v rozsahu platnosti této deklarace uplatnit při ošetření výjimky.

Je-li v deklaraci handleru uveden *identifikátor_výjimky*, musí být dříve deklarován v deklaraci výjimky. Handler se zavolá, pokud je tato výjimka aktivována příkazem `SIGNAL`. Stejná výjimka nesmí být uvedena dvakrát ve stejných nebo různých deklaracích handleru ve stejném rozsahu platnosti. Je-li v deklaraci této výjimky uveden *sqlstate*, pak handler je přiřazen i k tomuto *sqlstate*.

Je-li v deklaraci handleru uvedeno `SQLSTATE` *sqlstate*, pak handler se zavolá, kdykoli tento *sqlstate* nastane.

Je-li v deklaraci handleru uvedeno slovo `SQLEXCEPTION`, `SQLWARNING` nebo `NOT FOUND`, handler se zavolá, nastane-li libovolná chyba kromě čtení neexistujícího záznamu v příkazu `FETCH` (pro `SQLEXCEPTION`), libovolné varování (pro `SQLWARNING`) nebo čtení neexistujícího záznamu v příkazu `FETCH` (pro `NOT FOUND`). Výjimka *Čtení z neexistujícího záznamu* (*sqlstate* '02000') je výjimečná tím, že handler pro ní se musí deklarovat při každém použití konstrukce `LOOP...FETCH...END LOOP`, zřejmě z tohoto důvodu má tak výsadní postavení.

Nastane-li při provádění SQL příkazů výjimka, hledá se nejbližší handler pro tuto výjimku. Je-li nějaký handler nalezen, pak v závislosti na druhu handleru:

- `CONTINUE`: provede se handler a pokračuje se v provádění příkazů za příkazem, který způsobil výjimku;
- `EXIT`: provede se handler a pokračuje se v provádění příkazů za koncem složeného příkazu, který obsahuje deklaraci handleru;
- `UNDO`: provede se `ROLLBACK` do stavu při vstupu do složeného příkazu, poté se provede handler a pokračuje se v provádění příkazů za koncem složeného příkazu, který obsahuje deklaraci handleru;
- `REDO`: provede se `ROLLBACK` do stavu při vstupu do složeného příkazu, poté se provede handler a pokračuje se v provádění příkazů od začátku složeného příkazu, který obsahuje deklaraci handleru.

Handlery `REDO` a `UNDO` smějí být deklarovány pouze v atomickém složeném příkazu. Nelze v nich používat transakční příkazy.

Příklad použití:

při průchodu kurzoru je třeba testovat existenci dalšího řádku

```
DECLARE CONTINUE HANDLER FOR NOT FOUND BEGIN
    SET err_notfound=TRUE; END;
LabelLoop: LOOP
    FETCH NEXT FROM cur INTO TatoFirma, TatoHodnota;
    IF err_notfound IS TRUE THEN LEAVE LabelLoop; END IF;
...
```

Příkaz SIGNAL

```
příkaz_SIGNAL ::= SIGNAL identifikátor_výjimky;
```

Příkaz vyvolá zadanou uživatelem definovanou výjimku. Výjimka musí být předem deklarována. Je-li deklarován handlerpro tuto výjimku, handler se provede a v provádění příkazů se pokračuje v závislosti druhu handleru a místě, kde byl nalezen.

Pokud se pro zadanou výjimku nenajde handler na žádné úrovni, nastane chybový *sqlstate*45000.

Příkaz RESIGNAL

```
příkaz_RESIGNAL ::= RESIGNAL [ identifikátor_výjimky ];
```

Příkaz se smí použít pouze uvnitř handleru výjimky. Příkaz předá zadanou výjimku resp. výjimku, která je právě zpracovávána (není-li *identifikátor_výjimky* uveden), ke zpracování vnějšími handlery. Je-li vně prováděného handleru deklarován handler pro tuto výjimku, handler se provede a v provádění příkazů se pokračuje v závislosti na druhu handleru a místě, kde byl nalezen.

Pokud se pro zadanou výjimku nenajde handler na žádné úrovni, nastane chybový *sqlstate*45000.