

# Jemný úvod do informixu

Verze dokumentu: 1.0

1. května 2002



# Obsah

<b>1</b>	<b>Začínáme</b>	<b>11</b>
1.1	Kde co je, nastavení . . . . .	11
1.2	Další možné problémy . . . . .	11
1.3	Kdo co může . . . . .	12
1.4	Archivace . . . . .	12
1.5	Datové typy užívané v Informixu . . . . .	13
<b>2</b>	<b>Formuláře</b>	<b>14</b>
2.1	Nabídka form . . . . .	14
2.2	Vytvoření standardního formuláře . . . . .	14
2.2.1	Volba Query . . . . .	15
2.3	Sekce specifikace formuláře . . . . .	15
2.4	Sekce ATTRIBUTES . . . . .	17
2.4.1	COMMENTS . . . . .	18
2.4.2	DEFAULT . . . . .	18
2.4.3	FORMAT . . . . .	18
2.4.4	INCLUDE . . . . .	18
2.4.5	LOOKUP . . . . .	18
2.4.6	PICTURE . . . . .	19
2.5	Působnost atributů položek . . . . .	19
2.6	Vícetabulkový formulář a spojení tabulek . . . . .	19
2.7	Sekce INSTRUCTIONS . . . . .	20
2.8	Řídící bloky . . . . .	20
2.9	Víceobrazkový formulář . . . . .	22
2.10	Položka DISPLAYONLY . . . . .	22
2.11	Přehled editačních kláves modulu PERFORM . . . . .	24
2.12	Překlad a spuštění formuláře z příkazové řádky . . . . .	24
<b>3</b>	<b>Generátor sestav ACE</b>	<b>26</b>
3.1	Úvodní informace . . . . .	26
3.2	Nabídka REPORT a generování standartní sestavy . . . . .	26
3.3	Sekce specifikace sestav . . . . .	26
3.3.1	Sekce DATABASE . . . . .	27
3.3.2	Sekce DEFINE . . . . .	27
3.3.3	Sekce INPUT . . . . .	28
3.3.4	Sekce OUTPUT . . . . .	28
3.3.5	Sekce SELECT . . . . .	29
3.3.6	Sekce FORMAT . . . . .	29
3.4	Příkazy, funkce a operátory v řídících blocích . . . . .	30
3.4.1	PRINT . . . . .	30
3.4.2	COLUMN . . . . .	30
3.4.3	SPACE . . . . .	30

3.4.4	USING . . . . .	30
3.4.5	CLIPPED . . . . .	31
3.4.6	LINENO a PAGENO . . . . .	31
3.4.7	SKIP . . . . .	31
3.4.8	NEED . . . . .	31
3.4.9	PAUSE . . . . .	31
3.4.10	TODAY . . . . .	31
3.5	Řídící bloky . . . . .	32
3.5.1	Řídící blok ON EVERY ROW . . . . .	32
3.5.2	Řídící bloky FIRST PAGE HEADER a PAGE HEADER . . . . .	32
3.5.3	Řídící bloky PAGE TRAILER a ON LAST ROW . . . . .	32
3.5.4	Řídící bloky BEFORE GROUP OF a AFTER GROUP OF . . . . .	32
3.6	Další funkce a příkazy v řídících blocích . . . . .	33
3.6.1	Funkce pracující s datem a časem . . . . .	33
3.6.2	Agregační funkce . . . . .	33
3.6.3	Další funkce . . . . .	34
3.7	Řídící příkazy v řídících blocích sestav . . . . .	34
3.8	Překlad a spuštění sestavy z příkazové řádky . . . . .	35
3.9	Dva příklady na ukázkou . . . . .	35
<b>4</b>	<b>Systém uživatelských nabídek USER-MENU</b>	<b>36</b>
4.1	Přehled systému uživatelských nabídek . . . . .	36
4.2	Volba USER-MENU . . . . .	36
4.3	Specifikace uživatelské nabídky . . . . .	36
4.4	Návrh systému uživatelských nabídek . . . . .	37
4.5	Spuštění systému uživatelských nabídek . . . . .	37
<b>5</b>	<b>Informix-4GL</b>	<b>38</b>
5.1	Úvod . . . . .	38
5.1.1	Co to je? . . . . .	38
5.1.2	Produkty . . . . .	38
5.2	Překlad programu . . . . .	38
5.2.1	Překlad a spuštění v INFORMIX-4GL RDS . . . . .	38
5.2.2	Překlad a spuštění v INFORMIX-4GL Compiler . . . . .	38
5.3	Programátorské prostředí INFORMIX-4GL . . . . .	39
5.3.1	i4gl . . . . .	39
5.3.2	r4gl . . . . .	40
5.4	Struktura programu . . . . .	40
5.5	Základní prvky jazyka 4gl . . . . .	40
5.5.1	Definice proměnných . . . . .	40
5.5.2	Globální proměnné . . . . .	41
5.5.3	Přiřazovací příkaz . . . . .	41
5.5.4	Předdefinované konstanty . . . . .	41
5.5.5	Pole . . . . .	41

5.5.6	Záznamy . . . . .	41
5.6	Řízení toku programu . . . . .	42
5.6.1	Volání funkce nebo reportu . . . . .	42
5.6.2	Předávání dat funkci nebo rutině report . . . . .	42
5.6.3	Návrat hodnot z funkce do hlavního programu . . . . .	42
5.6.4	Cykly . . . . .	42
5.6.5	Návěští . . . . .	43
5.6.6	Příkaz IF . . . . .	43
5.6.7	Rozvětvení programu . . . . .	43
5.7	Práce s databázemi . . . . .	44
5.7.1	Výběr dat z databáze . . . . .	44
5.7.2	Scroll kurzory . . . . .	45
<b>6</b>	<b>4GL — menu, okna, formuláře</b>	<b>47</b>
6.1	Příkaz MENU . . . . .	47
6.2	Příkaz OPTIONS . . . . .	48
6.3	Nápověda . . . . .	48
6.4	Komunikace s uživatelem . . . . .	48
6.5	Formátování dat . . . . .	49
6.6	Práce s formuláři . . . . .	49
6.7	Obrazovkové záznamy . . . . .	50
6.8	Příkazy pro práci s formuláři . . . . .	50
6.9	Práce s okny a formuláři . . . . .	51
6.10	Formulářová pole . . . . .	51
6.11	Knihovni funkce . . . . .	53
<b>7</b>	<b>Optimalizace dotazů SQL v INFORMIXU</b>	<b>54</b>
7.1	Optimalizující techniky . . . . .	54
7.1.1	Porozumění celému systému . . . . .	55
7.1.2	Porozumění aplikaci . . . . .	55
7.1.3	Měření aplikace . . . . .	55
7.1.4	Nalezení vinných funkcí . . . . .	56
7.2	Optimizér . . . . .	56
7.2.1	Důležitost pořadí tabulek . . . . .	56
7.2.2	Join bez filtrů . . . . .	56
7.2.3	Join se sloupcovými filtry . . . . .	58
7.2.4	Plánování s použitím indexů . . . . .	60
7.2.5	Jak optimalizér pracuje . . . . .	61
7.2.6	Vstup optimalizéru . . . . .	61
7.2.7	Přístupový filtr . . . . .	62
7.2.8	Vybrání přístupové cesty . . . . .	63
7.2.9	Vybrání plánu dotazu . . . . .	63
7.2.10	Čtení plánu . . . . .	63
7.3	Časová cena dotazu . . . . .	64

7.3.1	Aktivity v paměti . . . . .	64
7.3.2	Řízení diskového přístupu . . . . .	64
7.3.3	Diskové bloky . . . . .	64
7.3.4	Vyrovnávací paměť stránek . . . . .	65
7.3.5	Cena čtení řádky . . . . .	65
7.3.6	Cena sekvenčního přístupu . . . . .	65
7.3.7	Cena nesequenčního přístupu . . . . .	65
7.3.8	Cena přístupu podle ID . . . . .	66
7.3.9	Cena indexového přístupu . . . . .	66
7.3.10	Nízká cena malých tabulek . . . . .	66
7.3.11	Cena síťového přístupu . . . . .	66
7.4	Rychlé dotazy . . . . .	67
7.4.1	Příprava testovacího prostředí . . . . .	67
7.4.2	Studium schématu . . . . .	67
7.4.3	Studium Dotazového plánu . . . . .	68
7.4.4	Modifikace dotazu . . . . .	68
7.4.5	Přepsání joinů prostřednictvím pohledů . . . . .	68
7.4.6	Vyhnout se nebo zjednodušit třídění . . . . .	68
7.4.7	Odstranění sekvenčního přístupu do velkých tabulek . . . . .	69
7.4.8	Užití sjednocení k vyhnutí se sekvenčnímu přístupu . . . . .	69
7.4.9	Nahrazení autoindexů indexy . . . . .	70
7.4.10	Užití smíšeného indexu . . . . .	70
7.4.11	Užití bcheck na podezřelé indexy . . . . .	70
7.4.12	Zrušení a přestavění indexu po UPDATE . . . . .	70
7.4.13	Vyhnout se vztažným poddotazům . . . . .	70
7.4.14	Vyhnout se obtížným regulárním výrazům . . . . .	71
7.4.15	Vyhnout se nevýchozím podřetězcům . . . . .	71
7.4.16	Užití dočasných tabulek k urychlení dotazů . . . . .	71
<b>8</b>	<b>Tvorba a užití uložených procedur</b>	<b>72</b>
8.1	Přehled . . . . .	72
8.2	Tvorba uložených procedur . . . . .	72
8.2.1	Metoda DB-Access . . . . .	72
8.3	Využití SQL API . . . . .	73
8.3.1	Komentář a dokumentace SP . . . . .	73
8.4	Diagnóza překladových chyb . . . . .	73
8.4.1	Metoda DB-Access . . . . .	73
8.4.2	Přístup pomocí SQL-API . . . . .	73
8.4.3	Způsob uložení procedury v databázi . . . . .	74
8.5	Spuštění procedury . . . . .	74
8.6	Odladování procedur . . . . .	75
8.6.1	Znovu zavedení procedury do DB . . . . .	75
8.7	Přístupová práva a SP . . . . .	75
8.7.1	Práva potřebná pro vytvoření SP . . . . .	76

8.7.2	Práva potřebná pro spuštění SP . . . . .	76
8.7.3	Typ SP s právy vlastníka . . . . .	76
8.7.4	Typ s právem DBA . . . . .	76
8.8	Proměnné a výrazy . . . . .	76
8.8.1	Podřetězce proměnných . . . . .	77
8.8.2	Nejvyšší prioritu mají definované proměnné . . . . .	77
8.8.3	Přiřazení hodnot proměnným . . . . .	77
8.9	Řízení běhu programu . . . . .	77
8.9.1	Větvení . . . . .	78
8.9.2	Cykly . . . . .	78
8.10	Zpracování funkcí . . . . .	78
8.10.1	Funkce OS . . . . .	78
8.10.2	Rekurze SP . . . . .	78
8.10.3	Víceřádkové návratové hodnoty . . . . .	78
8.11	Zpracování výjimek . . . . .	79
<b>9</b>	<b>INFORMIX SQL Triggers</b>	<b>80</b>
9.1	Úvod . . . . .	80
9.2	Vytvoření trigru . . . . .	80
9.3	Zrušení trigru . . . . .	81
9.4	Vyvolání trigru . . . . .	81
9.5	Omezení v trigrech . . . . .	81
9.6	Systémové tabulky o trigrech . . . . .	82
9.7	Příklady . . . . .	82
9.7.1	Příklad 1 . . . . .	82
9.7.2	Příklad 2 . . . . .	83
9.7.3	Příklad 3 . . . . .	83
9.7.4	Příklad 4 . . . . .	83
<b>10</b>	<b>Informix ESQL/C</b>	<b>85</b>
10.1	Co to je . . . . .	85
10.2	Výrazy pro preprocesor ESQL/C . . . . .	85
10.3	Host-proměnné . . . . .	86
10.3.1	Deklarace . . . . .	86
10.3.2	Rozsah platnosti . . . . .	86
10.3.3	Vztah mezi datovými typy C a SQL . . . . .	87
10.3.4	Pole . . . . .	87
10.3.5	Struktury . . . . .	87
10.3.6	Definice typů (typedef) . . . . .	88
10.3.7	Hodnota NULL . . . . .	88
10.3.8	Deklarace host-proměnných typu ukazatel na znak . . . . .	88
10.3.9	Deklarace host-proměnných jako parametry funkce . . . . .	88
10.4	Indikační proměnné (indicator variables) . . . . .	88
10.4.1	Deklarace . . . . .	89

10.4.2	Funkce . . . . .	89
10.4.3	Použití . . . . .	89
10.5	Ošetření chyb . . . . .	89
<b>11</b>	<b>Transakce v INFORMIXU</b>	<b>91</b>
11.1	Co je to transakce? . . . . .	91
11.2	Příkazy používané v souvislosti s transakcemi . . . . .	91
11.3	Databáze bez transakcí (databáze bez logu) . . . . .	91
11.4	Databáze s implicitními transakcemi . . . . .	91
11.4.1	MODE ANSI . . . . .	91
11.4.2	non MODE ANSI . . . . .	92
11.5	Omezení při používání transakcí . . . . .	92
11.6	Používání transakcí . . . . .	92
11.7	Akce u kterých není možné provést ROLLBACK . . . . .	92
<b>12</b>	<b>Pohledy v INFORMIXU</b>	<b>93</b>
12.1	Možná použití pohledů . . . . .	93
12.2	Vytvoření pohledu . . . . .	93
12.3	Omezení pro pohledy . . . . .	93
12.4	Chování pohledů . . . . .	94
12.5	Vliv změn v databázi na pohledy . . . . .	94
12.6	Zrušení pohledu . . . . .	94
<b>13</b>	<b>Práce s BLOB (Binary Large Objects)</b>	<b>95</b>
13.1	Datový typ . . . . .	95
13.2	Programování . . . . .	95
13.3	Políčka společná všem datovým umístěním . . . . .	96
13.4	BLOB v paměti . . . . .	97
13.4.1	Příklad: načtení blobu do paměti . . . . .	97
13.4.2	Příklad: Zapsání blobu z paměti . . . . .	98
13.5	Blob v otevřeném souboru udaném descriptoru . . . . .	98
13.6	Blob v souboru udaném jménem . . . . .	98
13.7	Blob v uživatelem definované oblasti . . . . .	99
13.7.1	Otevření lokátoru . . . . .	99
13.7.2	Zavření lokátoru . . . . .	99
13.7.3	Načtení lokátoru . . . . .	99
13.7.4	Zápis lokátoru . . . . .	99
<b>14</b>	<b>Fyzický a logický log v INFORMIXu</b>	<b>101</b>
14.1	Fyzický log . . . . .	101
14.2	Logický log . . . . .	101
14.3	Dlouhá transakce . . . . .	102



<b>15 OnLine</b>	<b>104</b>
15.1 Obnova dat . . . . .	104
15.2 Procedura obnovy dat . . . . .	104
15.3 Checkpointy . . . . .	106
15.4 Na co si dat pozor v prostredi OnLine . . . . .	106
<b>16 Utility</b>	<b>108</b>
16.1 Zpusob ulozeni dat na disku . . . . .	108
16.1.1 Root Dbspace, Temporary Dbspace . . . . .	108
16.1.2 Temporary Tables (explicit/implicit) . . . . .	109
16.2 Utility pro diskovou administraci . . . . .	109
16.2.1 oncheck . . . . .	109
16.2.2 oninit . . . . .	109
16.2.3 onload . . . . .	110
16.2.4 onlog . . . . .	110
16.2.5 onmode . . . . .	111
16.2.6 onparams . . . . .	112
16.2.7 onspaces . . . . .	113
16.2.8 onstat . . . . .	114
16.2.9 ontape . . . . .	115
16.2.10 onunload . . . . .	116
<b>17 Zámky</b>	<b>117</b>
17.1 Druhy zámeků: . . . . .	117
17.2 Oblast působnosti zámeků: . . . . .	117
17.2.1 Databáze . . . . .	117
17.2.2 Tabulka . . . . .	117
17.2.3 Řádka/stránka/klíč . . . . .	118
17.3 Doba trvání zámeků (závisí na tom, jestli se používají transakce) .	118
17.3.1 tabulka . . . . .	118
17.3.2 Řádek/stránka/klíč . . . . .	118
17.4 Izolační úrovně . . . . .	118
17.5 Lock mode . . . . .	119
17.6 Demonstrační programy . . . . .	119
<b>A Seznam autorů</b>	<b>120</b>
<b>B Formuláře</b>	<b>120</b>
<b>C Reporty</b>	<b>120</b>
<b>D Zámky</b>	<b>120</b>

## Úvod

Tento dokument vznikl z referátů přednesených na seminářích předmětu *Praktikum z INFORMIXu*, který probíhal v zimním semestru ve školním roce 1998 až 1999. Jeho verze nabyla hodnoty 1.0, čímž je řečeno, že jsou všechny referáty zpracovány. Pokud kdokoliv nalezne chybu, chce něco doplnit a podobně, budou skripta opravena nebo doplněna.

Při psaní v  $\text{\TeX}$ u nebyly použity žádné zvláštní vymoženosti. Tisk na jakémkoliv tiskárně by měl být zcela v pořádku.

Veškeré dotazy nechť směřují na mou poštovní schránku<sup>1</sup>.

Pokud chce někdo zasílat informace o nových verzích, nechť mi taktéž napíše.

Na tvorbě těchto skript jsem strávil již přes dvacet osm hodin svého života.

Tento dokument se nesnaží být typografickým skvostem už z toho důvodu, že jednotlivé kapitoly byly sebrány od různých autorů píšících různým stylem a nebylo v silách autora směřovat všechny styly k jednomu. Jde tedy o převážně studijní materiál.

**Upozornění:** Tato skripta vznikla z referátů, jež jsou chráněna autorským zákonem uvedených autorů. Skripta smějí být použita pouze a jen pro studijní účely. Skripta mohou být šířena pouze s dodanými ukázkovými soubory vážící se k jednotlivým kapitolám. Autor skript neodpovídá za chyby vzniklé ať v těchto skriptech (buť jen gramatických či typografických) nebo v důsledku používání návodů v těchto skriptech obsažených.

---

<sup>1</sup>kdo ji ještě nezná, zde je: rk@atrey.karlin.mff.cuni.cz

# 1 Začínáme

## 1.1 Kde co je, nastavení

Informix je na ulab-8 (nutno se připojit přes telnet). Na této stanici je server i klientské nástroje, klienti se serverem komunikují prostřednictvím sdílené paměti.

Základní komunikace probíhá pomocí klientských nástrojů isql a dbaccess zadáním příkazu isql nebo dbaccess ze shellu. Dojde k otevření standardních menu (isql viz [3] a manuály [1] resp. [5]; dbaccess je popisován v manuálu [2], ale protože jeho funkce jsou podmnožinou funkcí isql, stačí výše uvedené texty). Řadu činností těchto klientů lze spouštět též z příkazové řádky (tento způsob nebude na praktiku probírán).

Pro správné napojení je nutné mít dosazené hodnoty proměnných INFORMIXDIR a PATH (nejlépe v .profile):

```
INFORMIXDIR=/SPIS/DB/INFORMIX_NEW
export INFORMIXDIR
PATH=$PATH:$INFORMIXDIR/bin
export PATH
```

případné nastavení editoru: proměnná DBEDIT (nezapomenout na export, jinak Informix volá editor vi).

Pokud INFORMIXDIR není dosazeno, hlásí se „Unknown message number 32766.“

## 1.2 Další možné problémy

Další problémy spuštění mohou mít především tyto příčiny (častěji se vyskytuje spíše jen ta první):

- při rebootu ulab-8 se neinicializovala sdílená paměť — projeví se to tím, že se systém bez ohlášení chyby sám vrátí do shellu nebo v některých situacích hlásí chybu „Shared memory not initialized“: pomůže dát (z shellu, na ulab-8) příkaz **tbinit** a po novém promptu normálně začít pracovat (např. isql)
- pokud toto nepomůže, mohou být zcela zaplněné tzv. logické logy, je třeba je backupovat — to může provést jen administrátor Informixu (pošlete mi mail na [riha@ksi.ms.mff.cuni.cz](mailto:riha@ksi.ms.mff.cuni.cz) nebo zavolejte na (2191)4268)
- nevhodným použitím kill popř. po jiném zásahu mohl „zahynout“ démon Informixu (hlásí se něco jako „Daemon died“), přitom nebyly zrušeny semaforey — opět třeba uvědomit administrátora
- již příliš mnoho procesů pracuje s Informixem („Too many users“) — nutno počkat, až se někdo odpojí.

Naplnění logických logů, založení databáze, aktuálně pracující uživatele a řadu dalších věcí lze zkontrolovat použitím utility **tbmonitor** (opět se spouští na ulab-8 ze shellu).

### 1.3 Kdo co může

Klienty isql nebo dbaccess může spustit v podstatě každý, práva jsou nastavena až u databází („zakladatel“ má právo DBA, nižší jsou pak Resource a Connect — ta se udělují pomocí SQL příkazu Grant).

Nástroje isql popř. dbaccess umožňují mimo jiné založení (definici) databáze a tabulek, a to ve společném databázovém prostoru, proto každá databáze musí mít unikátní jméno.

Upozornění: „ostatní“ data týkající se databáze (SQL skripty, definice formulářů, exportovaná data atd.) se ukládají vždy do aktuálního adresáře uživatele, tj. do adresáře nastaveného před spuštěním např. isql, pro pořádek je vhodné si pro každou databázi založit zvláštní adresář.

Naplnění tabulky daty je nejvhodnější provádět interaktivně pomocí tzv. standardního formuláře (vytvořeného v isql), což je velmi blízké použití jazyka QBE.

Další možnost je dávkově pomocí SQL příkazu

```
LOAD FROM <soubor> [DELIMITER "<znak>"] INSERT INTO <tabulka>
```

(soubor musí být v takovém tvaru, jak by ho vytvořil z hotové tabulky příkaz

```
UNLOAD TO <soubor> [DELIMITER "<znak>"] SELECT * FROM <tabulka>
```

), nebo SQL příkazem

```
INSERT INTO <tabulka> VALUES (<seznam hodnot>)
```

, uloženým jako SQL skript.

### 1.4 Archivace

Protože je Informix na fakultě používán pouze k výukovým účelům, není prováděna pravidelná archivace dat systémovými prostředky. V tomto ohledu spoléháme na archivaci dat pro celou laboratoř, prováděnou kolegou Pavlem Semeřákem a dále na samostatnou archivaci uživatelů.

Utilita dbexport s parametrem <jm\_databáze> vytvoří v aktuálním adresáři uživatele podadresář <jm\_databáze>.exp a do něho uloží i skript definující schéma databáze (<jm\_databáze>.sql) i datové soubory (<tabulka>.unl) pro všechny tabulky.

Utilita dbimport <jm\_databáze> naopak provede vytvoření celé databáze včetně dat (předpokládá ovšem existenci výše uvedeného podadresáře .exp s příslušnými soubory).

Utilitou `dbschema` `[-t <tabulka>] -d <databaze> <soubor>` lze vytvořit „jen“ sql script definující danou databázi popř. tabulku; takový skript je pak nutno spouštět z `isql` nebo `dbaccess`.

## 1.5 Datové typy užívané v Informixu

[illegible]

## 2 Formuláře

Poznámka předem: je vhodné mít aktivní nějakou databázi.

Pokud chcete využít příkladu, se kterým jsem se to učil, pak v příloze B jsou poznámky k přiloženým příkladům.

Pro práci s formuláři slouží modul PERFORM, kterýžto jesti prostředkem INFORMIX-SQL. Umožňuje interaktivně navrhovat obrazovkové formuláře a pak pomocí nich manipulovat s daty.

Obrazovkové formuláře budeme krátce zvat **formuláře**. Někdy se též volají **maska**. **Specifikace formuláře** označuje popis struktury formuláře. Formuláře můžeme navrhovat nejen pomocí nabídek INFORMIX-SQL, ale i z příkazové řádky pomocí libovolného editoru. Je však třeba dodržet některá pravidla:

specifikace formuláře je v souboru s příponou **.per** přeložený formulář je v souboru **.frm**

### 2.1 Nabídka form

Po volbě nabídky Form z hlavního formuláře se objeví menu:

Run	Spustí formulář přechodem do nabídky PERFORM
Modify	Zavolá externí editor k editaci existující specifikace
Generate	Vygeneruje a přeloží standardní specifikace formuláře
New	Vytvoření nové specifikace formuláře
Compile	Přeloží specifikaci formuláře
Drop	Smaže specifikaci formuláře
Exit	Vrátí se do nabídky INFORMIX-SQL

### 2.2 Vytvoření standardního formuláře

Volbou Generate se objeví výzva ke vstupu jména formuláře. Zadejte libovolné jméno (libovolné znamená začátek písmenem a zbytek písmena, podtržítka a číslice). Po zadání jména následuje výběr tabulky, ke kteréžto vytváříme formulář. Potvrďte libovolnou a ještě potvrďte položku Table-selection-complete (volbou Select-more-tables byste pokračovali ve výběru tabulek výsledný formulář by byl vícetabulkový). Pokud vše proběhlo v pořádku, po vygenerování tabulky jste zpět v nabídce FORM. Zvolte Run a jméno formuláře, který jste právě vygenerovali. Zobrazí se následující nabídka:

Query	Vybere řádky vyhovující podmínkám do tzv. aktivní skupiny (active set) řádek a zobrazí první nalezenou. Zobrazenou řádku zoveme aktuální.
Next	Přesune se na další řádku aktivní skupiny a zobrazí ji.
Previous	Přesune se na předchozí řádku aktivní skupiny a zobrazí ji.
View	Umožní prohlížet datové typy BLOB.

Add	Přejde do formuláře a umožňuje zadávat nová data.
Update	Přejde do formuláře a umožňuje opravovat data.
Remove	Smaže aktuální řádku.
Table	Změní aktuální tabulku vícetabulkového formuláře.
Screen	Zobrazí další obrazovku formuláře.
Current	Aktualizuje právě zobrazenou řádku posledními údaji.
Master	Přejde na MASTER tabulku.
Detail	Přejde na DETAIL tabulku.
Output	Přesměrování výstupu.
Exit	Přejde do nadřazené nabídky.

### 2.2.1 Volba Query

Po zvolení Query si můžete vybrat, jaké řádky chcete mít v aktivní skupině. Pokud nezadáte žádný řetězec, zvolí se všechny, které jsou v tabulce již zadané. Výběr se provádí podle následujících pravidel:

>xxx	hodnoty větší než xxx
<xxx	hodnoty menší než xxx
<>xxx	hodnoty různé od xxx
>=xxx	hodnoty větší nebo rovné xxx
<=xxx	hodnoty menší nebo rovné xxx
=xxx	hodnoty rovné xxx
xxx:yyy	všechny hodnoty mezi xxx a yyy
xxx—yyy	hodnoty xxx nebo yyy
<<	minimální hodnotu sloupce
>>	maximální hodnotu sloupce
*	libovolný i prázdný řetězec
?	libovolný jeden znak
[XYZ]	X, Y nebo X

Příklad:

K*	vše s K na začátku
[Kk]*	vše co začíná na k nebo K
K??	začíná to na K a má to délku tři
=	řádky s hodnotou NULL

Výběr aktivní skupiny musí předcházet použití voleb Next, Previous, Updae, Remove, Current apod.

## 2.3 Sekce specifikace formuláře

Vraťte se do nabídky FORM a zvolte si volbu Modify. Zadejte jméno formuláře a pak editor. Specifikace se skládá z maximálně pěti sekcí:

DATABASE	Specifikuje jméno databáze, ke kterému formulář patří.
SCREEN	Určuje rozmístění <b>návěští</b> a <b>položek formuláře</b> na obrazovce. <b>Návěští</b> mohou obsahovat libovolný text (kromě hranatých závorek). <b>Položky formuláře</b> jsou ohraničeny hranatými závorkami. Určují oblast obrazovky, ve které se zobrazují a vstupují data. Uvnitř každé <b>položky formuláře</b> je uveden <b>identifikátor položky</b> — kombinace písmen a čísl. <b>Identifikátor položky</b> pouze spojuje <b>položku</b> se záznamem v sekci <b>ATTRIBUTES</b> . Sekce <b>SCREEN</b> se může opakovat.
TABLES	Definuje všechny tabulky používané formulářem.
ATTRIBUTES	Určuje typ zobrazovaných a vstupujících dat odkazem na databázový sloupec, spojuje sloupce tabulek, specifikuje různé atributy zobrazování a vstupu dat.
INSTRUCTIONS	Nepovinná sekce popisující speciální vlastnosti formuláře

Syntaxe:

```
DATABASE jméno_databáze END
```

```
SCREEN [SIZE řádky [BY sloupce]]
{
  návěští [id_položky1]
  návěští [id_položky2]
  .
  .
  .
  návěští [id_položkyN]
}
[END]
```

```
TABLES
  [jiné_jméno_tabulky] [vlastník.] tabulka
  ...
```

```
ATTRIBUTES
  id_položky1 = popis_položky;
  id_položky2 = popis_položky;
  ...
  id_položkyN = popis_položky;
[END]
```

```
[INSTRUCTIONS
  instrukce; ...]
```



[END]

## 2.4 Sekce ATTRIBUTES

V této sekci definujeme pro každou položku uvedenou v sekci SCREENS sloupce tabulky. Bylo-li např. v sekci SCREENS řádek:

```
jmeno          [f000          ]
```

pak v sekci attributes musí být minimálně následující řádek:

```
f000 = film.jmeno;
```

což znamená, že se odkazujeme na tabulku film a atribut jméno. Zobrazení můžeme ovlivnit následujícími pravidly:

**AUTONEXT:** po naplnění položky se cursor přesune na další

**COMMENTS**

**DEFAULT**

**DOWNSHIFT:** převádí velká písmena na malá

**FORMAT**

**INCLUDE**

**NOENTRY:** zabraňuje vstupu údaje do položky při vkládání nové řádky (ADD). Nezabraňuje však změně dat v této položce při opravě řádky.

**NOUPDATE** zabraňuje změně, nikoliv však vkládání

**PICTURE**

**PROGRAM**

**QUERYCLEAR**

**REQUIRED:** položka musí být zadána

**REVERSE:** zobrazuje data inverzně

**RIGHT:** určuje zarovnání znakových dat

**UPSHIFT:** převádí malá písmena na velká

**VERIFY:** vyžaduje opakovaný vstup dat do položky s kontrolou na shodu vstupu.

**WORDWRAP** [COMPRESS]

**ZEROFILL:** Určuje zarovnání číselných dat v položce doprava s doplněním vedoucích nul.

### 2.4.1 COMMENTS

COMMENTS = "zpráva"

Při přechodu kursoru do se zobrazí uvedený komentář.

### 2.4.2 DEFAULT

DEFAULT = hodnota

standardní hodnota položky, u času se může zadat TODAY

### 2.4.3 FORMAT

FORMAT = "formátovací řetězec"

Určuje výstupní formát hodnot typu DECIMAL, SMALLFLOAT, FLOAT a DATE. Číselné hodnoty můžete formátovat kombinací symbolů „#“, „.“ a „-“. Dále lze používat:

dd	reprezentuje číslo dne v měsíci
ddd	reprezentuje zkratku dne v týdnu
mm	reprezentuje číslo měsíce
mmm	reprezentuje zkratku měsíce
yy	reprezentuje dvouciferný rok
yyyy	reprezentuje čtyřciferný rok

Příklady:

FORMAT = "mm/dd/yy"

FORMAT = "yyyymmdd"

FORMAT = "dd-mm-yy"

FORMAT = "mmm dd, yy"

FORMAT = "#####.##"

FORMAT = "###,###.##"

### 2.4.4 INCLUDE

INCLUDE = (seznam hodnot)

Určuje seznam povolených hodnot pro danou položku formuláře. Po každém vstupu hodnoty do položek se kontroluje, zda je v seznamu uvedena. Členy seznamu uddělujte čárkami, znakové řetězce uvádějte v uvozovkách.

### 2.4.5 LOOKUP

LOOKUP = [id\_položky = [tabulka.]jméno\_sloupce, ...]

JOINING [\*][tabulka.]jméno\_sloupce

Zajišťuje **spojení s nahlédnutím**, které umožňuje prohlížet údaje z jiné tabulky. Na základě údajů v položce se vyberou data ze spojené tabulky. Tato data se pouze zobrazí, ale nelze s nimi pracovat, slouží pouze pro informaci.

Lze jej doplnit o **ověřované spojení**, které zabrání vložení údaje do tabulky v případě, že údaj neexistuje ve spojené tabulce (označené hvězdičkou).

#### 2.4.6 PICTURE

PICTURE = "šablona"

Formátuje zadávaný text. Šablona může obsahovat tyto speciální znaky:

A	reprezentuje písmeno
#	reprezentuje číslici
X	reprezentuje libovolný znak

Význam ostatních znaků zůstává zachován.

### 2.5 Působnost atributů položek

Působnost atributů položek formuláře na spojovací sloupce můžete řídit. Oddělení působnosti atributů na sloupce zajistíte středníkem:

```
f000 = kniha.isbn;  
      exemplar.isbn, NOENTRY, NOUPDATE;
```

NOENTRY a NOUPDATE v tomto případě platí pouze pro exemplář.isbn, nikoliv však pro kniha.isbn.

### 2.6 Vícetabulkový formulář a spojení tabulek

Při generování standardního formuláře můžete zadat více tabulek (volba **Select-more-tables**). Jednoduché spojení (simple join) se provede v sekce ATTRIBUTES podle syntaxe:

```
[*][tabulka.][sloupec[,atributy,...]] [=   
[*][tabulka.][sloupec[,atributy,...]]...
```

Příklad:

```
f000 = firma.idf = zamestnanec.idf (pozor, generovany zaznam v sekci ATTRI-  
BUTE pro položku zamestanec.idf musíme vymazat).
```

Někdy je potřeba zaručit, aby k řádkám zadávaným do druhé tabulky vždy existovala spojená řádka z první tabulky. Toto **ověřované spojení** (verify join) zajišťuje integritu dat databáze. **Ověřované spojení** zřídíme hvězdičkou u první tabulky.

```
f000 = *firma.idf = zamestnanec.idf  
Složené spojení o něco níže.
```

## 2.7 Sekce INSTRUCTIONS

Tato sekce je nepovinná. Umožňuje definovat nestandardní oddělovače položek (field delimiters), složená spojení (Composite Join), vazby dominantních a detailních tabulek (Relationship Master/Detail) a řídicí bloky (Control Blocks).

Standardní oddělovače jsou hranaté závorky. Předdefinování:

DELIMITERS "xy"

x je levý oddělovač a y je pravý oddělovač

**Složené spojení** (composite join) se zřizuje mezi tabulkami, které jsou spojeny více než jedním sloupcem. **Složené spojení** může být navíc **ověřované**.

Ke stanovení **složeného spojení** musíte nejprve zajistit jednoduché spojení mezi všemi sloupci participujícími na spojení tabulek. Pak doplňte do sekce **INSTRUCTIONS** instrukci **COMPOSITES**:

Syntaxe:

```
COMPOSITES<tabulka1.sloupec1,...,tabulka1.sloupecN>[*]  
          <tabulka2.sloupec1,...,tabulka2.sloupecN>
```

Dále lze doplnit spojení dvou tabulek vazbou **Master/Detail**. Zjednodušuje to hledání spojených záznamů tabulky. V nabídce **PERFORM** je tato vazba podporována volbami **Master** a **Detail**.

Volbu **Detail** lze použít k přechodu od řádky dominantní tabulky k spojeným řádkám detailní tabulky, které jsou vybrány do aktivní skupiny a kterými pak můžete listovat.

Volba **Master** slouží k přepnutí na spojenou řádku nadřazené tabulky.

Syntaxe:

master.tabulka MASTER\_OF detailní\_tabulka

Master i detail slouží k práci s více obrazovkovými formuláři. Volba Table přepíná pouze mezi tabulkami formuláře. Jméno aktuální tabulky lze nalézt v pravém horním rohu obrazovky. Volby Master a Detail umožňují přechody od řádek master tabulky ke spojeným řádkám detailní tabulky a naopak.

## 2.8 Řídicí bloky

Řídicí bloky se umožňují řídit provádění operací před a po přístupu k datům. Používají se v sekci INSTRUCTIONS. Řídicí bloky jsou prováděny na základě řady podmínek splněných před nebo po editaci (vstupu) hodnoty položky. Řídicí bloky můžete použít k těmto účelům:

- k nastavení kursoru do určité položky při zadávání nebo opravě řádky
- k výpočtům nad položkami formuláře a k zobrazení výsledku do jiné položky
- k zobrazení komentáře při splnění jistých podmínek
- k zobrazení dat ve speciální položce **DISPLAYONLY**

Syntaxe:

```
{BEFORE|AFTER}-{EDITADD|EDITUPDATE|REMOVE}...OF  
  {tabulka|sloupec}...  
akce
```

```
AFTER {ADD|UPDTE|QUERY|DISPLAY}  
OF tabulka  
akce  
...
```

Přehled řídicích bloků:

#### • BEFORE

- **EDITADD** *sloupec* **EDITUPDATE**: Akce se provede, jakmile kursor vstoupí do položky, ještě před tím, než vložíte nebo opravíte data
- **EDITADD** *tabulka* **EDITUPDATE**: Akce se provede před vložením nebo změnou dat
- **REMOVE** *tabulka*: Akce se provede poté, co se uživatel rozhodne vymazat řádku tabulky, ale ještě před tím, než se řádka skutečně vymaže

#### • AFTER

- **EDITADD** *sloupec* **EDITUPDATE**: Akce se provede poté, co kursor opustí položku, ale ještě před vstupem na položku další.
- **EDITADD** *tabulka* **EDITUPDATE**: Akce se provede po zadání a potvrzení všech dat (po stisku ESC), ale ještě před zápisem do databáze.
- **ADD** *tabulka*: Akce se provede po vložení a potvrzení všech dat a pozařazení řádky do tabulky.
- **DISPLAY** *tabulka*: Akce se provede po každé operaci modulu **PERFORM**, která zobrazí data na obrazovku.
- **QUERY** *tabulka*: Akce se provede po akci **Query**.

V řídicích blocích můžete použít následující instrukce:

- **LET** *id\_položky* = *hodnota*: Přiřazuje hodnoty určité položce formuláře (konstanty, jiný identifikátor položky, agregační funkce, aritmetické oprace).
- **COMMENTS** [**BELL**] [**REVERSE**] "**zpráva**": Zobrazuje uvedený komentář s případnými argumenty.
- **NEXFIELD** *id\_položky*—**EXITNOW** Posunuje kursor do uvedené položky nebo ukončí práci s formulářem.

- **ABORT** Ruší příkaz pro vložení, změnu nebo výmaz dat. Funguje jako stisk klávesy interrupt.
- **IF-THEN-ELSE** Možnost použití BEGIN...END. Kdo nezná pascal, má smůlu.

Příklady:

```
BEFORE REMOVE OF kniha
BEGIN
  COMMENTS BELL REVERSE "Raději nic mazat nebude"
  ABORT
END
```

## 2.9 Víceobrazkový formulář

Formulář je možno rozdělit na více obrazovek, přepíná se pak pomocí Screen v menu. Syntaxe:

```
SCREEN
{

}
SCREEN
{

}
SCREEN
{

}
END
```

## 2.10 Položka DISPLAYONLY

- Není spojena s žádným sloupcem z tabulky databáze
- Slouží k zobrazení určitých hodnot
- Hodnoty se přiřazují v sekci instrukcí
- Musí se explicitně přiřadit typ
- Možnost použití všech vyjma SERIAL
- Délka typu CHAR je určena velikostí položky v sekci SCREEN (rozsah mezi „[“ a „]“)

- Možnost použití: DEFAULT, FORMAT, REVERSE, UPSHIFT, DOWNSHIFT, QUERYCLEAR, RIGHT, ZEROFILL
- syntaxe: DISPLAYONLY TYPE datový\_typ [,atributy, ...]

Příklad:

```
SCREEN
{
...
Ferda Mravenec je: [txt1      ]
...
}

ATTRIBUTES
...
txt1 = DISPLAYONLY TYPE CHAR
...

INSTRUCTIONS
...
LET txt1 = "Uplny nemehlo"
...
```

## 2.11 Přehled editačních kláves modulu PERFORM

<b>CTRL-X</b>	maže jeden znak
<b>CTRL-A</b>	přepíná režim vkládání/přepisování (doteď nechápu, proč je standardní přepisování, to se museli u Informixů pěkně \censored{vožrat}).
<b>CTRL-D</b>	maže obsah položky od pozice kursoru do konce řádky
<b>CTRL-F</b>	posune se na další položku
<b>CTRL-B</b>	posune se na předcházející položku
<b>CTRL-C</b>	vymaže všechny položky
<b>CTRL-P</b>	opakuje poslední zadanou hodnotu položky
<b>CTRL-W</b>	help
<b>CTRL-I</b>	přesun na následující položku
<b>ESCAPE</b>	potvrdí vstup (jak neskutečně božské)
<b>DELETE</b>	přeruší vstup (nečekaně)
<b>RETURN</b> <b>CTRL-J</b> šipka dolů	následující položka
<b>CTRL-K</b> šipka nahoru	přesun na předchozí položku
<b>CTRL-H</b> šipka vlevo <b>BACKSPACE</b>	přesun po položce o znak vlevo
<b>CTRL-L</b> šipka vpravo	přesun po položce o znak vpravo

## 2.12 Překlad a spuštění formuláře z příkazové řádky

překlad: **sbformbld** *formular* (připomínám, že je nutno mít příponu .per)  
if (vše ok)

existuje soubor formular.frm  
else

view formular.err

opakuj process

Spuštění formuláře: **sperform** *formular*

Další možnost překladu:

**sbformbld -d**

Budete vyzváni k zadání:

- jména standardního formuláře
- jména databáze
- jména tabule, pro kterou se má formulář vytvořit



Vytvoří se specifikace formuláře pro zadané tabulky, uloží se do souboru a rovnou se přeloží.

## 3 Generátor sestav ACE

### 3.1 Úvodní informace

ACE je programový prostředek INFORMIX-SQL, který umožňuje prostřednictvím tzv. specifikace sestavy (report) uspořádat (vybírat a formátovat) výstupní informace z databáze. Může být použit například k vytvoření faktur, dopisů, jmenovek, pozvánek a podobně.

Specifikace sestavy — samostatný soubor sestava.**ace**

Přeložená specifikace: sestava.**arc**

### 3.2 Nabídka REPORT a generování standartní sestavy

Menu REPORT:

<b>RUN</b>	Provede sestavu (generuje tiskový výstup)
<b>MODIFY</b>	Editace existující sestavy
<b>GENERATE</b>	Vygeneruje a přeloží standartní specifikaci sestavy
<b>NEW</b>	Vytvoření (editace) zcela nové sestavy
<b>COMPILE</b>	Přeloží specifikaci sestavy
<b>DROP</b>	Zruší existující specifikaci sestavy
<b>EXIT</b>	Přechod do nadřazeného menu

Při vytváření nové sestavy se zadává:

- jméno reportu (sestavy)
- jméno databáze
- jméno tabulky (případně tabulek)

### 3.3 Sekce specifikace sestav

Specifikace sestavy se může skládat z šesti částí:

<b>DATABASE</b>	Určuje databázi, ze které se budou data vybírat
<b>DEFINE</b>	Definuje lokální proměnné sestavy a parametry příkazové řádky
<b>INPUT</b>	Zajišťuje interaktivní vstup parametrů
<b>OUTPUT</b>	Určuje okraje a délku vstupních stran a umístění výstupu
<b>SELECT</b>	Specifikuje příkaz(y) SELECT pro výběr dat
<b>FORMAT</b>	Formátuje výstupní data

- Povinné sekce:
  - **DATABASE**
  - **SELECT**
  - **FORMAT**
- nepovinné sekce:
  - **DEFINE**
  - **INPUT**
  - **OUTPUT**

Pořadí sekcí ve specifikaci sestav je závazné.

### 3.3.1 Sekce **DATABASE**

- Je povinná
- Musí být uvedena jako první
- Určuje databázi, ze které se budou data vybírat

Syntaxe:

```
DATABASE
jméno_databáze
END
```

### 3.3.2 Sekce **DEFINE**

- Nepovinná
- Za sekcí **DATABASE**
- Definuje lokální proměnné sestavy a parametry příkazové řádky

Syntaxe:

```
DEFINE
[VARIABLE jméno_proměnné datový_typ] ...
[PARAM [celé_číslo] jméno_proměnné datový_typ] ...
END
```

Jména proměnných

- Musí začínat písmenem
- Smí obsahovat písmena, číslice a podtržítko

- Smí být maximálně 18 znaků dlouhá

Parametry jsou naplněny hodnotami argumentů sestavy pouze při spuštění sestavy z příkazové řádky. Odkazy na hodnoty proměnných musí předcházet symbol \$.

### 3.3.3 Sekce **INPUT**

- Nepovinná
- Za sekcemi **DATABASE**, **DEFINE**
- Zajišťuje interaktivní vstup hodnot do lokálních proměnných

Syntaxe:

```
INPUT
  [PROMPT FOR jméno_proměnné USING "řetězec"]
END
```

Pomocí sekce **INPUT** můžete vytvářet sestavy, které interaktivně přebírají od uživatele parametry sestavy.

### 3.3.4 Sekce **OUTPUT**

- Nepovinná
- Za sekcemi **DATABASE**, **DEFINE**, **INPUT**
- Určuje okraje a délku vstupních stran a umístění výstupu

Syntaxe:

```
OUTPUT
  [REPORT TO {"jméno_souboru" | PIPE "program" | PRINTER}]
  [LEFT MARGIN celé_číslo]
  [RIGHT MARGIN celé_číslo]
  [TOP MARGIN celé_číslo]
  [BOTTOM MARGIN celé_číslo]
  [PAGE LENGTH celé_číslo]
END
```

Standartně:

- Výstup na obrazovku
- Levý okraj 6, pravý okraj 132, horní okraj 3, dolní okraj 3
- Délka stránky 66 řádek

### 3.3.5 Sekce **SELECT**

- Povinná
- Za sekcemi **DATABASE**, **DEFINE**, **INPUT**, **OUTPUT**
- Specifikuje příkaz(y) **SELECT** pro výběr dat
- Obsahuje jeden nebo více příkazů **SELECT**
- V příkazu **SELECT** můžete použít odkazy na proměnné a parametry, ale musí je předcházet symbol \$
- Za posledním příkazem **SELECT** je **END**

Syntaxe:

```
příkaz_SELECT; ...  
END
```

### 3.3.6 Sekce **FORMAT**

- Povinná
- Za sekcemi **DATABASE**, **DEFINE**, **INPUT**, **OUTPUT**, **SELECT**
- Formátuje výstupní data

V této sekci můžete uvést jeden ze dvou základních typů formátu:

a) **Jednoduchý formát** uspořádává data do sloupců nebo do posloupnosti po sobě jdoucích záznamů.

Syntaxe:

```
FORMAT  
    EVERY ROW  
END
```

b) **Složitější formát** se specifikuje se kombinací sedmi řídicích bloků

Syntaxe:

```
FORMAT  
    [PAGE HEADER řídicí_blok]  
    [PAGE TRAILER řídicí_blok]  
    [FIRST PAGE HEADER řídicí_blok]  
    [ON EVERY ROW řídicí_blok]  
    [ON LAST ROW řídicí_blok]  
    [BEFORE GROUP OF řídicí_blok]  
    [AFTER GROUP OF řídicí_blok]  
END
```

Za návěštím řídicího bloku (**PAGE HEADER**, **PAGE TRAILER**, ...) se uvádí posloupnost tiskových příkazů zajišťujících tisk řídicího bloku.

### 3.4 Příkazy, funkce a operátory v řídicích blocích

Můžeme zde používat:

- Tiskové příkazy
- Funkce
- Agregační funkce
- Operátory kterými určíte kde, co a jak tisknout

#### 3.4.1 PRINT

Tiskne znakové řetězce, hodnoty sloupců, obsahy proměnných a hodnoty funkcí uvedených v *seznamu\_výrazů*. Středník potlačí odřádkování.

Syntaxe:

```
PRINT [seznam_výrazů] [;]  
PRINT FILE "jméno_souboru"
```

Druhý řádek vytiskne obsah souboru.

#### 3.4.2 COLUMN

V seznamu\_výrazů příkazu **PRINT** převede aktuální pozici tisku v aktuální řádce do uvedeného sloupce.

Syntaxe:

```
COLUMN číselný_výraz
```

#### 3.4.3 SPACE

V seznamu\_výrazů příkazu **PRINT** přesune aktuální pozici tisku v aktuální řádce o uvedený počet mezer.

Syntaxe:

```
SPACE[S] číselný_výraz
```

#### 3.4.4 USING

Formátuje číselné a datové hodnoty *výrazu1* podle formátovacího řetězce *výraz2*.

Ve *výrazu2* můžete použít speciální znaky:

<	Zarovnává doleva
#	Zarovnává doprava
\$	Vpředu doplňuje symbolem měny
&	Na plnou délku doplňuje nulami

Syntaxe:

```
výraz1 USING výraz2
```

### 3.4.5 CLIPPED

Vynechává ze znakových dat koncové mezery.

Syntaxe:

znakový\_výraz CLIPPED

### 3.4.6 LINENO a PAGENO

Vrací aktuální číslo řádky a stránky.

Syntaxe:

LINENO

PAGENO

### 3.4.7 SKIP

Přeskočí uvedený počet řádek nebo konec strany.

Syntaxe:

SKIP číselný\_výraz LINES

SKIP TO TOP OF PAGE

### 3.4.8 NEED

Zajistí nerozdělené vytištění uvedeného počtu řádek následujících za příkazem **NEED**. Nevejde-li se uvedený počet řádek do konce strany, převede se tisková pozice na novou stranu.

Syntaxe:

NEED číselný\_výraz LINES

### 3.4.9 PAUSE

Vytiskne výzvu a čeká na potvrzení. Vhodné na konec strany.

Syntaxe:

PAUSE ["Výraz"]

### 3.4.10 TODAY

Vrací hodnotu typu **DATE** s aktuálním kalendářním datem. Hodnota typu **DATE** se standartně tiskne ve formátu mm/dd/yy. Můžete ji však přeformátovat operátorem **USING** a těmito speciálními symboly:

dd	Den v měsíci
ddd	Reprezentuje třípísmennou zkratku dne v týdnu
mm	Reprezentuje číselné vyjádření měsíce v roce
mmm	Reprezentuje třípísmennou zkratku měsíce v roce
yy	Zastupuje dvouciferné vyjádření roku
yyyy	Zastupuje čtyřciferné vyjádření roku

Syntaxe:

TODAY

### 3.5 Řídící bloky

#### 3.5.1 Řídící blok ON EVERY ROW

- Určuje zpracování jednotlivých řádek vrácených příkazem **SELECT**
- Za klíčovými slovy **ON EVERY ROW** uveďte tiskové příkazy pro tisk jedné řádky

#### 3.5.2 Řídící bloky FIRST PAGE HEADER a PAGE HEADER

- Tisk titulních stran dokumentů a horních záhlaví stran
- Blok **FIRST PAGE HEADER** je proveden pouze jednou, **PAGE HEADER** na začátku každé strany
- **FIRST PAGE HEADER** potlačí **PAGE HEADER** (na první stránce)

#### 3.5.3 Řídící bloky PAGE TRAILER a ON LAST ROW

- Vytisknutí patiček, závěrů, souhrnů
- Blok **ON LAST ROW** je proveden pouze jednou (na úplném konci sestavy), **PAGE TRAILER** na konci každé strany
- **ON LAST ROW** potlačí **PAGE TRAILER** (na poslední stránce)
- Na konec řídícího bloku **PAGE TRAILER** se při výstupu na obrazovku hodí příkaz **PAUSE**

#### 3.5.4 Řídící bloky BEFORE GROUP OF a AFTER GROUP OF

- Umožňují seskupovat data podle stejné hodnoty v uvedených sloupcích
- Blok **BEFORE GROUP OF** se provádí před zpracováním řádek skupiny



- Blok **AFTER GROUP OF** se provádí po zpracování řádek skupiny
- Mohou obsahovat stejné příkazy jako řídicí blok **ON EVERY ROW** a navíc se v nich mohou vyhodnocovat agregační funkce nad řádkami skupiny

### 3.6 Další funkce a příkazy v řídicích blocích

#### 3.6.1 Funkce pracující s datem a časem

Očekávají jako argument hodnotu typu DATE:

<b>DAY</b>	Vrací den v měsíci (1 – 31)
<b>WEEKDAY</b>	Vrací den v týdnu (1 – 7)
<b>MONTH</b>	Vrací měsíc v roce (1 – 12)
<b>YEAR</b>	Vrací rok

#### 3.6.2 Agregační funkce

Provádí výpočty nad všemi řádkami a nad řádkami skupin.

- **COUNT** Vrací počet všech řádek vrácených příkazem SELECT
- **PERCENT** Vrací procentuelní vyčíslení počtu řádků skupiny vzhledem k celkovému počtu vybraných řádek
- **TOTAL** Vrací součet hodnot sloupce ze všech vybraných řádek
- **AVERAGE (AVG)** Vrací průměrnou hodnotu sloupce ze všech vybraných řádek
- **MIN, MAX** Vrací minimální (maximální) hodnotu sloupce ze všech vybraných řádek

Agregační fce se také často používají v řídicích blocích skupin. Předchází-li je klíčové slovo **GROUP**, pak vracejí hodnoty získané výpočtem nad řádkami skupiny.

Syntaxe:

```
[GROUP] {COUNT | PERCENT} [WHERE podmínka]
[GROUP] {TOTAL | AVG | MIN | MAX} OF výraz
[WHERE podmínka]
```

### 3.6.3 Další funkce

- **DATE** Vrací řetězec s aktuálním kalendářním datem nebo datem určeným argumentem
- **MDY** Sestavuje kalendářní datum z položek data
- **TIME** Vrací aktuální čas

Syntaxe:

DATE

DATE (datový\_výraz)

MDY (číselný\_výraz, číselný\_výraz, číselný\_výraz)

TIME

**ASCII** Vrací znak s daným ASCII kódem (vhodné např. pro předání řídicích znaků na terminál nebo tiskárnu).

Syntaxe:

ASCII číselný\_výraz

## 3.7 Řídicí příkazy v řídicích blocích sestav

V řídicích blocích sestav můžete použít podmíněný příkaz **IF-THEN-ELSE**, cykly **WHILE** a **FOR** a přiřazovací příkaz **LET**. Do příkazů **IF-THEN-ELSE**, **WHILE** a **FOR** se začleňují buď jednotlivé příkazy nebo skupiny příkazů uzavřené do bloku **BEGIN-END**. Příkaz **IF-THEN-ELSE** musí mít v řídicích blocích **FIRST PAGE HEADER** a **PAGE HEADER** stejný počet vytištěných řádek v části **IF** i **ELSE**.

Syntaxe:

IF výraz THEN příkaz  
[ELSE příkaz]

WHILE výraz DO

FOR čítač = výraz1 TO výraz2 [STEP výraz3]

LET proměnná [číselný\_výraz [, číselný\_výraz]] = seznam\_výrazů

BEGIN příkaz ... END

### 3.8 Překlad a spuštění sestavy z příkazové řádky

Při návrhu sestav nemusíte nutně používat prostředí **INFORMIX-SQL**. Specifikaci sestavy můžete vytvořit libovolným editorem z příkazové řádky. Pouze je třeba zajistit, aby soubor se specifikací sestavy měl příponu **.ace**.

Překlad specifikace sestavy provedete příkazem

**saceprep** *jméno\_sestavy.ace*

Jestliže byl překlad úspěšný, pak je přeložená specifikace sestavy uložena v souboru *jméno\_sestavy.arc*

Nebyl-li překlad úspěšný, pak v souboru *jméno\_sestavy.err* najdete lokalizaci a popis chyby.

Úspěšně přeloženou specifikaci sestavy spustíte příkazem

**sacego** *jméno\_sestavy*

Tento způsob spouštění sestav musíte zvolit, chcete-li do sestavy předávat parametry na příkazové řádce.

### 3.9 Dva příklady na ukázkou

Informace o příkladech naleznete v příloze C.

## 4 Systém uživatelských nabídek USER-MENU

### 4.1 Přehled systému uživatelských nabídek

Systém uživatelských nabídek se skládá z jednotlivých nabídek a jejich voleb. Volby nabídky mohou provádět příkazy operačního systému, spouštět příkazové soubory dotazovacího jazyka, volat sestavy nebo formuláře, nebo přejít do další nabídky. Volby se vybírají šipkami a potvrzují klávesou Enter, nebo je lze volit odesláním jejich čísla. Návrat do nadřazené nabídky se provede klávesou e.

Při založení systému uživatelských nabídek se vytvoří v databázi dvě uživatelské tabulky, které uchovávají popis nabídek a voleb. Jmenují se sysmenus a sysmenuitems.

### 4.2 Volba USER-MENU

V hlavní nabídce INFORMIX-SQL je třeba zvolit User-menu. Pokud tabulky sysmenus a sysmenuitems ještě neexistují, vytvoří se a pak se objeví nabídka pro práci s user-menu.

- **Run** Spustí uživatelskou nabídku
- **Modify** Umožní opravit existující nabídku

### 4.3 Specifikace uživatelské nabídky

Spouští se klávesou m nebo volbou modify. Aktivuje modul PERFORM se společným formulářem tabulek sysmenus a sysmenuitems pro zadání nabídek. Horní část formuláře MENU ENTRY FORM obsahuje položky pro tabulku sysmenus, dolní část SELECTION SECTION pro tabulku sysmenuitems. Formulář obsahuje tyto položky:

- **Menu Name** Obsahuje jméno nabídky. Toto jméno musí být v systému jednoznačné, smí obsahovat písmena, číslice a podtržítka. Nabídka nejvyšší úrovně musí mít jméno main.
- **Menu Title** Obsahuje záhlaví nabídky. Je umístěno na první řádce na začátku nabídky. Je omezeno pouze délkou, která se vejde do formuláře.
- **Selection Number** Obsahuje číslo volby, které bude zobrazeno nalevo od volby v zobrazené nabídce. Jeho odesláním se příslušná volba aktivuje.
- **Selection Type** Určuje typ akce, kterou volba provede. Typ akce může být jeden z:
  - **F** Spustí formulář
  - **R** Spustí sestavu

- **M** Zavolá další nabídku
  - **Q** Provede příkazový soubor
  - **P** Provede příkaz operačního systému
  - **S** Vyvolá textem specifikované akce
- **Selection Text** Určuje text volby, který se objeví v nabídce vpravo od čísla volby. Omezen je pouze délkou položky. Pokud se volby nevejdou na jeden řádek na obrazovce, budou uspořádány do dvou sloupců.
  - **Selection Action** Určuje prováděnou akci volby podle zadaného typu volby. Je-li zadán typ Q a akce volby nebyla specifikována, přejde do editoru SQL. Je-li zadán typ volby R nebo F a akce nebyla specifikována, přejde do modulů ACE a PERFORM.

#### 4.4 Návrh systému uživatelských nabídek

Vytvoříme systém uživatelských nabídek, který má hlavní nabídku main a sedm voleb:

- **Dotazy** Nabídne vytvořené dotazy.
- **Dotazovací jazyk** Přejde do SQL editoru.
- **Formulare** Nabídne spuštění vytvořených formulářů.
- **PERFORM** Přejde do hlavní nabídky modulu PERFORM.
- **Sestavy** Nabídne spuštění vytvořených sestav.
- **ACE** Přejde do hlavní nabídky modulu ACE.
- **Datum** Volá systémový příkaz date.

Volby Dotazy, Formuláře a Sestavy budou mít ještě nabídky dotazy, fomulare a sestavy.

#### 4.5 Spuštění systému uživatelských nabídek

Systém uživatelských nabídek lze spustit

- V prostředí INFORMIX-SQL volbou User-menu, pak volbou Run
- V příkazové řádce příkazem isql dbkomix -u, nebo také isql dbkomix -u jméno\_nabídky

Standardně se objevuje nabídka main. Nastavením proměnné prostředí DB-MENU lze nastavit jinou počáteční úroveň nabídek nebo zpřístupnit pouze některou část uživatelské nabídky.

## 5 Informix-4GL

### 5.1 Úvod

#### 5.1.1 Co to je?

- 4GL = Fourth Generation Language, jazyk čtvrté generace
- Určen k vývoji db aplikací

#### 5.1.2 Produkty

- INFORMIX-4GL Compiler: překládá program do spustitelného kódu
- INFORMIX-4GL RDS (Rapid Development System): překládá program do pseudo kódu (p-code), který také interpretuje
- INFORMIX-4GL ID (Interactive Debugger): ladicí program

**Typické použití:** Vývoj a ladění pomocí INFORMIX-4GL RDS (rychlý překlad) + INFORMIX-4GL ID, výslednou aplikaci přeložit v INFORMIX-4GL Compiler (rychlý běh aplikace, protože se neinterpretuje).

### 5.2 Překlad programu

Překlad a spuštění programu pomocí INFORMIX-4GL RDS a INFORMIX-4GL Compiler se od sebe liší, proto jsou popsány samostatně.

#### 5.2.1 Překlad a spuštění v INFORMIX-4GL RDS

- pomocí fglpc přeložit jednotlivé moduly \*.4gl do souboru p-kódu \*.4go
- soubory \*.4go zkonkaténovat (např. cat) do jediného souboru něco.4gi (všechny přípony jsou nepovinné)
- programy p-kódu interpretuje fgldo

#### 5.2.2 Překlad a spuštění v INFORMIX-4GL Compiler

1. převod z 4GL do ESQL/C (**fglc**), povinná přípona **.ec**
2. převod z ESQL/C do C (**fglc2**), povinná přípona **.c**
3. převod z C do object kódu (lib. překladač jazyka **C**, např. **cc**), povinná přípona **.o**
4. linkování object modulů (lib. odpovídající linker, např. **ld**), lib. přípona, typicky **.4ge**

- z příkazové řádky lze to vše příkazem shellu:

```
c4gl [-e] [argumenty_cc] [-o cil]
      zdroj.4gl [ostatni.4gl] [ostatni.ec ...]
      [ostatni.c ...] [ostatni.o ...] [knihovny ...]
```

- volba -e způsobí překlad pouze do .c (.c se už nepřekládá a nelinkuje se)
- vyskytne-li se chyba, vytvoří se soubor .err s popisem chyby

### 5.3 Programátorské prostředí INFORMIX-4GL

Dvě vývojová prostředí (s malými rozdíly):

- **i4gl** — pro INFORMIX-4GL Compiler
- **r4gl** — pro INFORMIX-4GL RDS

#### 5.3.1 i4gl

- Module
  - New, Modify: volá externí editor pro editaci 4GL kódu
  - Compile: přeloží modul 4GL do souboru .4ge
  - Program-Compile: přeloží program definovaný ve volbě Program hlavního menu
  - Run: spustí přeložený program
- Program: určuje moduly, knihovny; specifikace programu jsou vždy uloženy v db syspgm4gl
  - New, Modify:
    - \* 4GL: zadává do 4gl Source jména modulů v kódu 4GL (bez přípony) a do 4gl Source Path adresáře, kde jsou moduly uloženy
    - \* Other: jako 4GL ale o modulech v kódu ESQ/L/C a C a object modulů
    - \* Libraries: jména uživatelských knihoven
    - \* Compile.Options: přepínače pro překladač jazyka C
  - Compile: přeloží celý program, uvažuje čas poslední změny (překládá jen to, co se od posledního překladu změnilo)
  - Planned.Compile: vypíše postup překladu, aniž by se překlad spustil

### 5.3.2 r4gl

V nabídkách Module a Program je navíc volba Debug, volající INFORMIX-4GL ID (debugger).

## 5.4 Struktura programu

- **program** se skládá z **modulů**
- v jednotlivých modulech jsou definovány **funkce** (MAIN, FUNCTION, REPORT)
- MAIN
  - v programu musí být vždy právě jedna tato funkce ukončená END MAIN
- FUNCTION
  - obecné funkce
  - ukončeno END FUNCTION
- REPORT
  - fce pro generování tiskových výstupů (sestav) umožňující tisk dat v nejrůznějších formátech. Uvnitř těchto funkcí lze užít příkazy, které jsou jinde nepřístupné
  - ukončeno END REPORT

## 5.5 Základní prvky jazyka 4gl

### 5.5.1 Definice proměnných

DEFINE prom1 TYP, prom2 TYP, ...

Příklady :

- DEFINE cislo INTEGER
- definice proměnné vazbou na sloupec tabulky. Proměnná získá typ sloupce. Před touto definicí se musí otevřít databáze s tabulkou příkazem DATABASE

DEFINE jmeno LIKE zamestnanec.jmeno



### 5.5.2 Globální proměnné

Definice proměnných s globální platností

```
GLOBALS
DEFINE globalni CHAR (20)
END GLOBALS
```

Proměnné mohou být definovány v libovolném programovém bloku, vždy má přednost lokální proměnná před globální.

### 5.5.3 Přiřazovací příkaz

```
LET promenna=hodnota
```

```
LET a=1
LET ret="abcd"
```

### 5.5.4 Předdefinované konstanty

1. TRUE
2. FALSE
3. NULL — prázdná hodnota, lze ji přiřadit každému typu
4. NOTFOUND — konstanta, která se vrátí při nenalezení dat např. příkazem SELECT

### 5.5.5 Pole

- DEFINE jmeno\_pole ARRAY [delka1,delka2,delka3] OF typ
- Pole mohou být maximálně třírozměrná.
- Pole deseti tříznakových řetězců: DEFINE pole ARRAY [10] OF CHAR (3)
- Přístup k položkám: LET pole[2]="XXX"

### 5.5.6 Záznamy

```
DEFINE osoba RECORD
jmeno CHAR(20),
vek INTEGER
END RECORD
```

Nebo opět definované odkazem na tabulku v databázi (opět musí být databáze otevřena):

```
DEFINE zamest RECORD LIKE zamestnanci.*
```

## 5.6 Řízení toku programu

### 5.6.1 Volání funkce nebo reportu

CALL jmeno\_funkce

### 5.6.2 Předávání dat funkci nebo rutině report

- data se předávají pouze hodnotou

CALL fce (par1,par2,par3)

```
FUNCTION fce (p1,p2,p3)
DEFINE p1 INTEGER,
p2 CHAR (20),
p3 INTEGER
....
END FUNCTION
```

### 5.6.3 Návrát hodnot z funkce do hlavního programu

CALL fce (par1,par2,par3) RETURNING navrat1,navrat2

```
FUNCTION fce (p1,p2,p3)
DEFINE p1 INTEGER,
p2 CHAR (20),
p3 INTEGER
....
RETURN p1, p2
END FUNCTION
```

- RETURN — okamžitý návrat do volajícího programu (stejný návrat provede i END FUNCTION a END REPORT)
- EXIT PROGRAM — okamžité ukončení programu (jinak program skončí po END MAIN)

### 5.6.4 Cykly

- FOR cyklus

```
FOR i= zacatek TO konec STEP krok
....
END FOR
```

– EXIT FOR — předčasné ukončení cyklu

– CONTINUE FOR — pokračování v cyklu

- WHILE cyklus

```
WHILE podminka
.....
END WHILE
```

– EXIT WHILE, CONTINUE WHILE - podobně jako u for cyklu

#### 5.6.5 Návěští

```
GOTO label1
LABEL label1: .....
```

#### 5.6.6 Příkaz IF

```
IF podminka THEN ...
ELSE ...
END IF
```

#### 5.6.7 Rozvětvení programu

```
CASE
WHEN podminka1
....
WHEN podminka2
....
OTHERWISE
....
END CASE
```

Druhý způsob (testuje se proměnná na příslušnou hodnotu):

```
CASE (promenna)
WHEN "A"
.....
WHEN "B"
.....
OTHERWISE
....
END CASE
```

## 5.7 Práce s databázemi

Ve 4gl lze používat příkazy pro práci s databázemi, tabulkami, sloupci, pro výběr dat z tabulek a podobně.

- *CREATE DATABASE jmeno* vytvoří novou prázdnou databázi (jméno může být uloženo ve znakové proměnné)
- *CLOSE DATABASE jmeno* uzavření databáze. Když se neuvede jméno, uzavřou se všechny databáze.
- *DROP DATABASE jmeno* zruší databázi (musí být uzavřena)
- *CREATE TABLE jmeno\_tabulky (sloupec1 TYP, sloupec2 TYP, ...)* vytvoření tabulky
- *DROP TABLE jmeno\_tabulky* zrušení tabulky
- *DATABASE jmeno* otevření databáze
- a spousta dalších

### 5.7.1 Výběr dat z databáze

- pomocí příkazu *SELECT*
- pomocí kurzoru pro víceřádkové výstupy

Samostatný příkaz *SELECT* lze použít jen tehdy, když návratová hodnota je pouze jedna (je vybrán jen jeden řádek). Jestliže je výstup víceřádkový, program zhavaruje. Pro víceřádkové výstupy je nutno použít kurzor (viz dále)

Příklad na příkaz *SELECT*:

```
DATABASE zamestnanci
DEFINE z_cislo LIKE zam.cislo
SELECT cislo INTO z_cislo FROM zam WHERE ...
DISPLAY "Cislo zamestnance:", z_cislo
```

*SELECT DISTINCT ...* nebo *SELECT UNIQUE ...* nebude brát v úvahu duplicitní řádky

Při víceřádkových výstupech je nutné použít kurzory. Postup při programování kurzoru je následující:

- deklarace kurzoru (*DECLARE*)
- otevření kurzoru (*OPEN*)
- načtení dat (*FETCH*)

- uzavření kurzoru (CLOSE)

Příklad:

```

DATABASE lide
DEFINE zam RECORD LIKE zamestnanci.*
DECLARE kurzor CURSOR FOR SELECT * FROM zamestnanci
OPEN kurzor
FETCH kurzor INTO zam.*
DISPLAY zam.*
CLOSE kurzor

```

Příkaz SELECT nalezne všechny řádky odpovídající dotazu a každý příkaz FETCH zpřístupní jeden řádek. Po příkazu FETCH se kurzor přesune na řádek následující.

Rozpoznání posledního řádku

- pomocí testování proměnné STATUS. Jestliže se příkaz FETCH nepodaří, je STATUS nastaven na předdefinovanou konstantu NOTFOUND.

### 5.7.2 Scroll kurzory

Použijí se tehdy, když chceme vyselectovanými řádky procházet v jiném pořadí než od prvního do posledního.

```

DECLARE kurzor SCROLL CURSOR FOR SELECT .....

```

A pak lze použít:

```

FETCH pohyb_kurzoru jmeno_kurzoru INTO ....

```

V sekci pohyb\_kurzoru lze uvést:

- FIRST - skok na první řádek
- LAST - skok na poslední řádek
- PREVIOUS - předchozí řádek
- NEXT - následující řádek
- CURRENT - kurzor zůstane na řádku
- RELATIVE +/- n - posun nahoru/dolů o n řádků relativně k aktuálnímu řádku
- ABSOLUTE n - skok na n-tý řádek

### **Elegantnější průchod více řádky najednou**

Příkaz `FOREACH` — projde všechny vybrané řádky a automaticky otevře a zavře kurzor

```
DECLARE kurzor CURSOR FOR SELECT ....  
FOREACH kurzor INTO radek.*  
.....  
další příkazy  
END FOREACH
```

Lze použít také `EXIT FOREACH` (předčasné ukončení) nebo `CONTINUE FOREACH` (pokračování)

## 6 4GL — menu, okna, formuláře

### 6.1 Příkaz MENU

Příkaz **MENU** se používá pro vytvoření nabídky.

Syntaxe:

```
MENU jmeno_nabidky
{BEFORE MENU| COMMAND {KEY (seznam_klaves)|
[KEY (seznam_klaves)] volba_nabidky [popis_nabidky]
[HELP cislo_napovedy]}}
příkaz
...
[CONTINUE MENU]
...
[EXIT MENU]
...
[NEXT OPTION volba_nabidky]
...
[SHOW OPTION {volba_nabidky | ALL}]
...
[HIDE OPTION {volba_nabidky | ALL}]
...
END MENU
```

**BEFORE MENU:** Příkazy, které se mají provést před zahájením práce s nabídkou.

*seznam\_klaves:* Znaky, funkční klávesy, řídicí klávesy (Ctrl), escape znaky, klávesy přerušení; nedoporučuje se používat klávesy Ctrl+A, Ctrl+D, Ctrl+H, Ctrl+J, Ctrl+K, Ctrl+L, Ctrl+M, Ctrl+Q, Ctrl+R, Ctrl+S, Ctrl+X.

**HELP** *cislo\_napovedy:* Zobrazení podrobnější nápovědy k volbě.

**CONTINUE MENU:** Přeskočí se příkazy, které jsou zde uvedeny.

**SHOW OPTION:** Zobrazení voleb.

**HIDE OPTION:** Skrytí voleb.

**NEXT OPTION:** Určení aktivní volby po této volbě.

**EXIT MENU:** Ukončení práce s nabídkou a přechod na příkaz za END MENU.

Pokud za COMMAND KEY není uvedena nabídka, volba nebude v nabídce vidět.

Nápověda je jednořádková zpráva, která se objeví, když je volba zvýrazněna, a která tuto volbu popisuje. Její maximální délka je 80 znaků.

## 6.2 Příkaz OPTIONS

Příkaz **OPTIONS** umožňuje v programu měnit standardně nastavené hodnoty některých parametrů. Klauzule:

**MENU LINE:** řádka, na níž bude zobrazena nabídka (standardně první řádka obrazovky nebo okna)

**PROMPT LINE:** řádka, na níž se budou zobrazovat zprávy pro operátora a odkud se bude přebírat jeho odpověď (standardně první řádka)

**MESSAGE LINE:** řádka, na níž se budou zobrazovat zprávy (standardně druhá řádka)

**COMMENT LINE:** řádka, na níž se budou zobrazovat komentáře formuláře (standardně předposlední řádka)

**ERROR LINE:** řádka pro zobrazování chyb (standardně poslední řádka)

**HELP FILE:** soubor s nápovědou

**HELP KEY** — klavesa pro zobrazení nápovědy. Standardně se používá Ctrl+W, nedoporučuje se používat některé kombinace kláves — viz příkaz **MENU**, seznam\_klaves.

Číslo řádek lze zadávat buď jako celá čísla, nebo pomocí klíčových slov **FIRST**, **LAST**, **FIRST+číslo**, **LAST-číslo**.

## 6.3 Nápověda

Nápověda se přiřazuje v příkazu **MENU** v klauzuli **HELP** pomocí čísla, které jednoznačně určuje nápovědu v souboru nápověd. Toto číslo se v souboru nápověd píše na řádce před vlastní nápovědou a píše se před ním tečka. Lze použít i více souborů nápověd, ale jen jeden soubor může být aktivní. Který to je, lze určit příkazem **OPTIONS**.

Hotový soubor s nápovědou se musí přeložit následujícím příkazem:

**mkmessage zdrojovy\_soubor cilovy\_soubor**

## 6.4 Komunikace s uživatelem

Příkaz **ERROR** zobrazí chybovou zprávu. Tato zpráva zmizí použitím dalšího obrazovkového příkazu. Syntaxe:

**ERROR seznam\_zprav [ATTRIBUTE (seznam\_atributu)]**

- *seznam\_zprav* — proměnné a řetězce
- *seznam\_atributu* — implicitní nastavení: inverzní zobrazení a pípnutí



- WHITE, YELLOW, MAGENTA, RED, CYAN, GREEN, BLUE, BLACK, REVERSE, BLINK, UNDERLINE, INVISIBLE, DIM, BOLD, NORMAL

Příkaz MESSAGE zobrazí zprávu. Nezmizí sama, je nutno ji zrušit. Syntaxe: MESSAGE seznam\_zprav [ATTRIBUTE (seznam\_atributu)]

- *seznam\_zprav* — proměnné a řetězce
- *seznam\_atributu* — viz seznam\_atributu u příkazu ERROR

Příkaz PROMPT slouží k jednoduchému vstupu dat. Po zadání dat zpráva zmizí. Syntaxe: PROMPT seznam\_zprav FOR [CHAR] promenna [HELP cislo\_napovedy]

- Vstup se uloží do proměnné.
- Klíčové slovo CHAR způsobí, že vstup bude přečten ihned bez čekání na Enter.
- Pokud skutečný vstup neodpovídá očekávanému, zobrazí se chybová hláška. Lze ji odstranit testováním proměnné STATUS a příkazem WHENEVER.

## 6.5 Formátování dat

- CLIPPED odstraní koncové mezery z proměnné typu CHAR.
- USING určí formát dat nebo numerického pole. Syntaxi má stejnou jako v INFORMIX-SQL.
- TODAY vrací systémové datum typu DATE.
- DATE převede datum z formátu DATE na znakový řetězec ve formátu daném klauzulí USING.

## 6.6 Práce s formuláři

Srovnání formulářů INFORMIX-SQL a INFORMIX-4GL:

- Ve 4GL se bude ignorovat LOOKUP, NOUPDATE, QUERYCLEAR, RIGHT, ZEROFILL a spojení tabulek.
- V sekci INSTRUCTION je platný pouze příkaz DELIMITERS a deklarace obrazovkových záznamů.
- Nelze používat víceobrazkové formuláře.
- Položky typu DISPLAYONLY jsou nahrazeny typem FORMONLY. Umožňují vstup i zobrazení dat.

- Formulář se opět vytvoří z hlavní nabídky příkazem FORM, přeloží se ale příkazem:

**form4gl -d**

Tento příkaz sám požádá o jméno formuláře, přidá příponu **.per** a zeptá se na jméno databáze a jména tabulek, nad kterými bude formulář vytvořen. Vytvořený soubor lze editovat. Existující formulář se přeloží příkazem:

**form4gl jmeno\_formulare**

nebo

**form4gl -v jmeno\_formulare**

Volba **-v** ověří, zda délky položek v obrazovkové části formuláře odpovídají délkám sloupců v databázi. Případné chyby se opět zaznamenávají do souboru *jmeno\_formulare.err*. Pokud byl překlad úspěšný, vytvoří se soubor *jmeno\_formulare.frm*.

## 6.7 Obrazkové záznamy

- Deklarují se v sekci INSTRUCTIONS klíčovými slovy SCREEN RECORD. Pomocí SCREEN RECORD se adresují položky nebo skupiny položek ve formuláři, pak se na ně lze odkazovat z kódu. Příklad:

```
SCREEN RECORD fir_obr1 (idf, ico,  nazev)
```

```
SCREEN RECORD fir_obr2 (idf THRU  nazev)
```

```
SCREEN RECORD firzam_obr (firma.idf,
                           firma.nazev,
                           zamestnanec.idz,
                           zamestnanec.jmeno,
                           zamestnanec.prijmeni)
```

- Smějí obsahovat položky z více tabulek.
- Pro každou tabulku uvedenou ve formuláři je vytvořen standardní obrazkový záznam, který obsahuje všechny sloupce dané tabulky použité ve formuláři a má stejné jméno jako tabulka.
- Přes obrazkové záznamy se pomocí příkazu INPUT a DISPLAY provádí vstup a výstup ve formulářích.

## 6.8 Příkazy pro práci s formuláři

- Příkaz OPEN FORM přiřadí formuláři interní jméno, kterým se na něj odkazují příkazy programu. Syntaxe: *OPEN FORM jmeno\_formulare FROM "soubor\_s\_formularem"* — jméno souboru se uvádí bez přípony **.frm**

- Příkaz `DISPLAY FORM` zobrazí formulář. Syntaxe: *DISPLAY FORM jmeno\_formulare*

## 6.9 Práce s okny a formuláři

- Příkaz `OPEN WINDOW` otevře okno daných rozměrů, na dané pozici a s danými atributy.  
Syntaxe:

```
OPEN WINDOW jmeno_okna AT radka, sloupec
WITH {cislo ROWS, cislo COLUMNS |
FORM "soubor_s_formularem"}
[ATTRIBUTE (seznam_atributu)]
```

Pokud je v okně jen jeden formulář, lze použít příkaz `OPEN WINDOW ... WITH FORM ...`

- Příkaz `CLEAR SCREEN` smaže obrazovku.
- Příkaz `CLEAR FORM` vymaže obsahy položek v daném formuláři.  
Syntaxe: *CLEAR FORM seznam\_polozek*  
Příkaz `CLEAR WINDOW` smaže vnitřek okna, nechá rámeček.  
Syntaxe: *CLEAR WINDOW jmeno\_okna*
- Příkaz `CLOSE` uvolní paměť zabranou při vytvoření formuláře.  
Syntaxe: *CLEAR FORM jmeno\_formulare*
- Příkaz `CLOSE WINDOW` zavře okno, zruší jeho atributy a obnoví obsah obrazovky pod oknem.  
Syntaxe: *CLOSE WINDOW jmeno\_okna*
- Příkaz `CURRENT WINDOW IS` označí okno jako aktuální.

## 6.10 Formulářová pole

Deklarace obrazovkového záznamu se provádí pomocí `SCREEN RECORD`, určuje se zde i počet prvků.

Tok dat mezi obrazovkovým a programovým polem zajišťují příkazy `DISPLAY ARRAY` a `INPUT ARRAY`.

`INPUT ARRAY` i `DISPLAY ARRAY` se ukončí klávesou Escape nebo Del.

Po překročení hranic obrazovkového pole text roluje, po překročení hranic programového pole se objeví chybové hlášení a terminál pípne.

Syntaxe:

```

INPUT ARRAY programove_pole [WITHOUT DEFAULTS]
FROM obrazovkove_pole.*
[HELP cislo_napovedy]
[{BEFORE {ROW | FIELD seznam_polozek | INSERT | DELETE}
[...] | AFTER {ROW | FIELD seznam_polozek |
INSERT | DELETE | INPUT} [...] | ON KEY (seznam_klaves)}]
přikaz
...
[NEXT FIELD {jmeno_polozky | NEXT | PREVIOUS}]
...
[EXIT INPUT]
...
END INPUT]

```

Klauzule:

- BEFORE ROW — po přesunu kurzoru na novou řádku, ale před vkládáním dat; má přednost před
- BEFORE INSERT
- BEFORE INSERT — před vložením nové řádky
- BEFORE DELETE — před smazáním řádky
- BEFORE FIELD — ihned po posunu do uvedené položky formuláře
- ON KEY — po stisku uvedené klávesy
- AFTER FIELD — při opouštění položky formuláře
- AFTER INSERT — po kompletním vložení nové řádky do pole; má přednost před AFTER ROW
- AFTER DELETE — po smazání řádky
- AFTER ROW — při opouštění řádky
- AFTER INPUT — po stisku Escape

Syntaxe:

```

DISPLAY ARRAY programove_pole TO obrazovkove_pole.*
[ATTRIBUTE (seznam atributu)]
DISPLAY ARRAY programove_pole TO obrazovkove_pole.*
[ATTRIBUTE (seznam atributu)]
{ON KEY (seznam klaves)}
přikaz

```

```
...  
[EXIT DISPLAY]  
...  
END DISPLAY}
```

Před provedením DISPLAY ARRAY je potřeba zavolat funkci SET\_COUNT(výraz) kvůli nastavení počtu naplněných řádek v programovém poli.

### 6.11 Knihovní funkce

- SCR\_LINE() vrací číslo aktuální řádky obrazovkového pole.
- ARR\_CURR() vrací číslo aktuální řádky programového pole.
- ARR\_COUNT() vrací celkový počet vyplněných řádek programového pole.
- SET\_COUNT() nastaví počet vyplněných řádek v programovém poli.

## 7 Optimalizace dotazů SQL v INFORMIXU

Jak dlouho bude trvat vyhodnocení vašeho dotazu? Kolik diskových operací během jeho zpracování bude potřebovat? Pro mnoho dotazů nezáleží, o kolik je uděláte rychleji než člověk, ale musí být provedeny v určitém limitu časovém nebo co zvládne váš stroj. Tento referát by měl být o zlepšování vašich dotazů. To předpokládá, že máte danou pevnou strukturu tabulek a nemůžete ji měnit.

Měl bych pokrýt následující témata:

- Obecná diskuse technik pro optimalizaci SW zdůrazňující všechny věci před zásahem do SQL výrazu
- Popis optimizéru, části rozhodující, jak bude daný dotaz proveden. Když víte, jak pracuje, můžete tvořit lepší dotazy.
- Diskuse o rychlostech operací
- Souhrn technik, které by vám měly pomoci zlepšit vaše dotazy.

### 7.1 Optimalizující techniky

Před tím, než začnete optimalizovat svoje SQL dotazy, je důležité udělat krok zpět a vidět je jako část většího systému skládajícího se z následujících částí:

- Jeden nebo více programů (Uložené dotazy, zkompilované obrazovkové formuláře a reporty a programy v jednom nebo více jazycích, v kterých je SQL vestavěno.)
- Jeden nebo víc počítačů (Přinejmenším jeden počítač, kde jsou uloženy programy a databáze.)
- Jeden nebo více správců (Lidé zodpovědní za programy.)
- Jeden nebo více uživatelů
- Jeden nebo více organizací

Můžete pracovat ve velké organizaci, kde je každá tato část oddělená, nebo si to můžete dělat vše sami na malém počítači.

V každém případě je důležité rozlišovat dva body o systému jako celku.

1. cíl systému je sloužit uživateli.
2. SQL výrazy tvoří pouze malou část systému, často se nejefektivnější zá-sahy do systému týkají jiné části.

Následující část je obecný postup pro analýzu problému výkonnosti. Smysl postupu je ukázat možná, i netechnická, řešení.

### 7.1.1 Porozumění celému systému

Uvažujte celý systém. Řešení může být ve změně plánování nebo řízení zdrojů

### 7.1.2 Porozumění aplikaci

Naučte se vše o porozumění aplikaci jako celku. Databázová aplikace má obvykle mnoho částí- uložené dotazy,obrazovkové formuláře, reporty, programy. Uvažujte je dohromady a ptejte se:

- Jak to funguje (Odhaluje zbytečné činnosti, zda každý krok akce je důležitý.)
- Proč to funguje (Zvláště ve starých aplikacích jsou nejasné kroky.)
- Pro koho to pracuje (Ujistěte se, že každý výstup někdo potřebuje a jestli mu nestačí jednodušší formát)
- Jaké části zdržují

### 7.1.3 Měření aplikace

Nemůžete dělat významné pokroky v problému, pokud ho nezměříte. Musíte najít způsob jak opakovaně změřit pomalé části aplikace. Důvody jsou takovéto:

- Bez čísel nemůžete popsat problém uživatelům a organizaci. Když všechno, co můžete říci je „je to příliš pomalé“, nebo je tam příliš mnoho I/O operací, moc tím nevyjádříte. Získat spolupráci uživatelů a podporu managementu vyžaduje být schopen říci: „Report běží 2h. 38 min a tak nemůžeme končit před 11“, nebo „Průměrný update trvá 13 sec, ale ve špičkách jsem naměřil 49.“.
- Bez čísel nemůžete dosáhnout významných pokroků, nemůžete vyjádřit číselně míru zrychlení a vytyčit si jasné cíle.
- Bez čísel nemůžete vyjádřit své pokroky

Čísla potřebujete pro změření malých úspěchů a pro rozhodování mezi alternativními možnostmi.

Př: Zjišťování času ve 4gl

```
DEFINE start_time Datetime HOUR TO FRACTION(2) ??
```

```
Elapsed INTERVAL MINUTE (4) TO FRACTION(2)
```

```
LET start_time=EXTEND(CURRENT,HOUR TO FRACTION(2))
```

{měřená část}

```
LET elapsed= EXTEND(CURRENT, HOUR TO FRACTION(2))-start_time
```

```
DISPLAY "Elapsed time was", elapsed
```

Poznámka: CURRENT dává čas normální, pokud chceme skutečný čas běhu, jsou na to C funkce, které můžeme volat z 4gl programů a ACE reportů.

#### 7.1.4 Nalezení vinných funkcí

Jedním z lepších zvyků je, že ve většině programů je malá část kódu (20 % nebo méně) měří čas běhu jednotlivých částí programu.

## 7.2 Optimizér

Optimizér je část databáze, která rozhoduje jak provádět dotazy. Jeho nejdůležitější práce je rozhodovat pořadí tabulek. K provedení takového rozhodnutí se rozhoduje k nejefektivnějšímu přístupu k jednotlivým tabulkám (sekvenčně, přes index, přes dočasný index) a musí rozhodnout, kolik řádků každé tabulky se bude podílet na konečné odpovědi.

Návrh optimizéru není vědecký a optimizér je částí databáze, která se pořád vyvíjí a zlepšuje.

My bereme v potaz verzi 4.1 a starší.

#### 7.2.1 Důležitost pořadí tabulek

Pořadí tabulek má velký vliv na rychlost joinů, což nejlépe uvidíme na příkladech.

#### 7.2.2 Join bez filtrů

```
SELECT C.customer_num, O.order_num, sum(items.total_price)
FROM customer C, orders o, items I
WHERE C.customer_num=O.customer_num
AND O.order_num=I.order_num
GROUP BY C.customer_num, O.order_num
```

Představte si tabulky bez indexů. Bez indexů se nemá DBE jinou volbu, než jednoduché hnížděné cykly, takže by vyhodnocování vypadalo nějak takto:

For each row in the customer table do:

    Read row into C



```

For each row in the orders table do:

    Read row into O

    If O.customer_num=C.customer_num then
        let Sum=0

        For each row in the items table do:

            Read row into I

            If I.order_num=O.order_num then

                Let Sum=Sum+I.total_price

            End If

        End For

        Prepare an output
        row from C,I, and
        Sum
    End If

End For

End For

```

Celkem to tedy vypadá

Všechny řádky v tabulce	Načteny
Customer	customer
Orders	customer * orders
Items	customer odp * orders odp * items

Toto není jediný možný plán dotazu. Jiný může pouze přehodit role customer a orders.

Číslo řádku v každé tabulce je vzato z systémové katalogové tabulky systables. Ve skutečnosti můžeme napsat dotaz počítající počet řádků, který může být použit v dotazovém plánu výše uvedeného:

```

SELECT C.nrows+C.nrows*O.nrows+O.nrows*I.nrows
FROM systables C, systables O, systables I
WHERE C.tabname="customer"
AND O.tabname="orders"
AND I.tabname="items"

```

Takto v podstatě optimizer předvídá rozsah práce, který vyžaduje daný plán. Řádkové účty v systables jsou měněny pouze když je vyvolán příkaz UPDATE STATISTIC, takže optimizér občas počítá se zastaralými informacemi.

### 7.2.3 Join se sloupcovými filtry

V předchozím případě nebyl vidět rozdíl v rozsahu práce dvou možných plánů. Účast sloupcového filtru to mění. Sloupcový filtr je WHERE výraz, který redukuje počet řádek, kterými tabulky přisluívají. Toto je předchozí dotaz s filtrem:

```
SELECT C.customer_num, O.order_num,sum(items.total_price)
FROM customer C,orders o, items I
WHERE C.customer_num=O.customer_num
AND O.order_num=I.order_num
AND O.paid_date IS NULL
GROUP BY C.customer_num, O.order_num
```

Výraz O.paid\_date IS NULL vyfiltruje některé řádky, které budou použity z tabulky orders. Jako předtím jsou zde dva plány. Plán začínající orders:

For each row in the orders table do:

    Read row into O

    If O.paid\_date is null then

        For each row in the customer table do:

            Read row into C

            If O.customer\_num=C.customer\_num then

                let Sum=0

                For each row in the items table do:

                    Read row into I

                    If I.order\_num=O.order\_num then

                        Let Sum=Sum+I.total\_price

                    End If

                End For

```

        Prepare an output row from C,I ,and Sum
    End If
End For
End If
End For

```

Nechť pdnull bude počet řádků, které určí filtr:

```
SELECT COUNT(*) FROM orders WHERE paid_date IS NULL
```

Pak to bude vypadat:

Všechny řádky v tabulce	Načteny
Orders	Orders
Customer	customer *pdnull
Items	customer * items *pdnull

Alternativní plán by vypadal:

```

For each row in the customer table do:
    Read row into C
    For each row in the orders table do:
        Read row into O
        If O.paid_date is null
            AND
            O.customer_num=C.customer_num
        then
            let Sum=0
            For each row in the items table do:
                Read row into I
                If I.order_num=O.order_num then

```

```

        Let Sum=Sum+I.total_price

    End If

End For

Prepare an output
row from C,I ,and
Sum

End If

End For

End For

Všechny řádky v tabulce      Načteny
Customer                     |customer|
Orders                       |customer|*|orders|
Items                        |customer |*|items|*pdnull|

```

Tyto dva plány mají stejný výsledek, přestože se jejich postup liší. Tento rozdíl však může představovat tisíce diskových přístupů.

#### 7.2.4 Plánování s použitím indexů

Předchozí případy nepoužívali indexy. To bylo nerealistické. Skoro všechny tabulky mají jeden nebo více indexů, které se projevují při plánování.

S indexy by plán vypadal takto:

```

For each row in the customer table do:

    Read row into C

    Look up C.customer_num in index on orders.customer_num

    For each matching row in the orders table do:

        Read row into O

        If O.paid_date is null then

            let Sum=0

            look up O.order_num in index on items.order_num

```

```

    For each maching row in the items table do:

        Read row into I

        Let Sum=Sum+I.total_price

    End For

    Prepare an output row from C,I ,and Sum

End If

End For

End For

```

To už je velká redukce.

Ale každý plán užívající indexy musí index načíst. Je obtížné předvídat, jak mnoho stránek indexu bude třeba načíst.

Fyzické uložení tabulky může také ovlivnit cenu užití indexu. Předchozí dotaz načítá customery v fyzickém pořadí.

Když je tedy výstup v pořadí customer\_number, budou se záznamy slévat, takže budou načítány pouze jednou. Pokud je fyzické uložení náhodné, může vést vyzvedání každé položky na novou stránku, což bude také dost zpomalovat.

### 7.2.5 Jak optimalizér pracuje

Optimizéry od 4.0 výše formulují všechny možné dotazovací plány. Pro každý plán odhadnou počet zkoušených řádků tabulek, načtených diskových bloků a (když je nainstalovaný INFORMIX-STAR) síťových přístupů, který mohou očekávat. Vyberou plán s nejmenší cenou.

### 7.2.6 Vstup optimizéru

Optimizér se řídí systémovými katalogy a jinými souhrnnými informacemi. Není zde čas pro dotazy typu SELECT COUNT(\*) k zjištění počtu řádků v tabulce. Informace přístupné optimizéru pocházejí ze systémových katalogových tabulek. Tyto zahrnují informace o:

- počtu řádků v tabulce (v době posledního UPDATE STATISTIC)
- jestli je sloupec omezený jako unikátní
- indexy

- které sloupce
- vzestupně/sestupně
- clustery

Katalogy podporované INFORMIX-ONLINE nahrazují doplňkový vstup:

- počet diskových stránek obsazených řádkami dat
- hloubku indexu B+ stromu (míra množství práce potřebná k nahlédnutí do indexu)
- počet diskových bloků obsazených indexem
- počet unikátních položek v indexu (dělených počtem řádků-pro odhad kolik řádků bude souhlasit s daným klíčem)
- druhá největší a druhá nejmenší hodnota v indexovaném sloupci (snaží se vyhodit extrémní hodnoty, optimizér předpokládá, že hodnoty jsou rovnoměrně rozloženy mezi těmito)

#### 7.2.7 Přístupový filtr

Optimizér nejdříve zkouší výrazy ve WHERE a kouká se po filtru. Oceňuje zvlášť pro každý filtr. Selektivita je číslo mezi 0 a 1, které značí část řádků, o kterých si optimizér myslí, že projdou. Hodně vybíravý filtr dostane 0, filtr málo filtrující dostane 1.

Optimizér si tedy poznamenává tyto fakta o dotazu

- Jestli je vhodný index. Když jsou vybírány jenom indexované sloupce, je rychlejší číst pouze indexové stránky a ne celé řádky (To je často možné v poddotazech.).
- Jestli index může být použit k hodnocení filtru. Pro tento účel je za indexovaný sloupec považován ten, co má index nebo ten, který je v nějakém složeném indexu na prvním místě. Zde můžeme uvažovat různé možnosti:
- Když je sloupec porovnáván se znakem, hostitelskou proměnou, nebo zvláštním poddotazem, nástroj bude moci užít index v průběhu načítání řádek
- Když je indexovaný sloupec porovnáván se sloupcem v jiné tabulce(join)
- Jestli index může být použit v ORDER BY (třídění)
- Jestli může být index využit v GROUP BY

### 7.2.8 Vybrání přístupové cesty

Pak vybere optimizér nejlepší způsob přístupu do tabulek z dotazu. Má tři možnosti

- sekvenčně
- podle indexu
- vytvořit dočasný index a použít ho

Volba mezi prvními dvěma způsoby závisí na tom, kolik odfiltrují filtry. Když je dostatečný filtr na indexovaném sloupci, přistupuje se pomocí indexu.

Když zde není filtr, musí přechít všechny řádky. Je obvykle rychlejší jednoduše číst řádky než čtení řádků přes index. Když se však řádky očekávají seřazené, můžeme ušetřit přímým čtením méně než řazením.

Dočasný index vytváří ve dvou případech

- Když žádná tabulka v joinu nemá na spojovaném sloupci index a tabulky jsou dostatečně velké, může se optimizér rozhodnout, že je rychlejší vystavět index na tabulku, než číst celou tabulku pro každý řádek druhé.
- Můžete také použít dočasný index pro tříděné nebo groupované sekvence. (Alternativa je dočasná tabulka, která se pak setřídí.)

### 7.2.9 Vybrání plánu dotazu

Se všemi těmito dostupnými informacemi, optimizér generuje všechny možné dotazy pro joinování tabulek v páru. Potom, když jsou joinovány více než 2 tabulky, užije optimizér nejlepší dvoutabulkové plány k připojení dvou tabulek ke třetí, třech ke čtvrté a tak dál.

Optimizér přidá cenu koncových prací pro kompletní joinovací plány. (ORDER BY, GROUP BY)

Konečně optimizér vybere nejmenší náročný plán a předá ho hlavní části k zpracování.

### 7.2.10 Čtení plánu

Volba optimizéru nemusí být tajemstvím. Můžete si vyhledat přesně, co je plán. Zapněte volbu SET EXPLAIN ON před tím, než vykonáváte dotaz. Na počátku dalšího dotazu optimizér vypíše vysvětlení jeho plánu do konkrétního souboru (jeho jméno a umístění závisí na operačním systému)(sqexplain.out). Ceny vypisuje v jednotlivých přístupech na disk, jiné akce jsou na to přepočítávány.

## 7.3 Časová cena dotazu

Nejvíce času enginovi zabírají při provádění dotazů dvě věci

- čtení dat
- porovnávání sloupcových hodnot s jinými.

Z těchto je čtení daleko pomalejší.

Tato část bude o tom, kde engin tráví čas, bude prozkoumávat vztahy pro tvorbu rychlejších dotazů.

### 7.3.1 Aktivity v paměti

Engin může s daty pracovat pouze v paměti. Musí je načíst do paměti před jejich porovnáváním s filtry. Musí načíst řádky z obou tabulek, než začne testovat joinovací podmínku. Engin si připravuje výstup v paměti konstrukcí vybraných sloupců z jiných řádků v paměti.

Většina z těchto aktivit jsou velmi rychlé. V závislosti na počítači engin může provést stovky nebo dokonce tisíce porovnání za sekundu. Čas strávený prací v paměti je obvykle zlomkem z celého času provedení dotazu.

Dvě aktivity v paměti mohou trvat dlouhou dobu.

- třídění
- porovnávání pomocí LIKE a MATCHES.

Některé typy dotazů, zvláště testy jeden nebo více znaků před nebo uprostřed hodnoty jsou dost drahé.

### 7.3.2 Řízení diskového přístupu

Trvá déle číst řádky z disku než testovat řádky přímo v paměti. Hlavním cílem optimizéru je redukovat data, která musí být čtena z disku, ale může eliminovat pouze nejvíce zřejmé neefektivitu.

### 7.3.3 Diskové bloky

Engin rozděluje disk na bloky pevné délky. Stejná velikost je používána pro všechny databáze řízené jedním enginem. Indexy jsou také ukládány v blocích.

V INFORMIX-ONLINE je velikost bloku nastavena při inicializaci. (Obvykle 4 kB). U jiných enginů INFORMIXU, které využívají cizí souborový operační systém, velikost bloku závisí na něm (typicky 1kB). Je možné definovat tabulky široké jako blok (některé enginy dovolují řádkům přesahovat velikost bloku). Typická velikost řádku je mezi 50 a 200 B, takže typický blok obsahuje 5-50 řádků. Indexová položka obsahuje klíč a 4Btový pointer, takže blok indexu typicky obsahuje 50-500 položek.



#### 7.3.4 Vyrovnávací paměť stránek

Engine má kus paměti vyhrazen pro kopie diskových bloků, které byly v poslední době čteny. Zůstává v naději, že tyto stránky budou znovu potřeba. Když se jeho očekávání splní, nemusí znovu přistupovat na disk.

Podobně jako u bloků, počet vyrovnávacích pamětí závisí na enginovi a souborovém operačním systému.

#### 7.3.5 Cena čtení řádky

Když si engin potřebuje otestovat řádek, musí ho přečíst z disku. Při tom načte jeden řádek, ale blok, který ho obsahuje (Když je řádek delší než blok, přečte tolik bloků, kolik je nezbytné.). Cena přečtení bloku je základní jednotka práce, kterou optimizér užívá pro své výpočty. Skutečná cena načtení bloku je proměnná a těžko předvídatelná. Je to kombinace následujících faktorů:

- Vyrovnávací paměti
- Přístupu více aplikací ke stejnému bloku
- Seek time
- Reakční čas

Cena načtení bloku může kolísat od mikrosekund (když je ve vyrovnávací paměti) přes několika milisekund do stovek milisekund.

#### 7.3.6 Cena sekvenčního přístupu

Disková cena je nejmenší, když se čte v pořadí, v kterém jsou data uložena na disku (každý blok načten pouze 1x).

Pokud je engin jediný program na disku, cena seeku je také minimalizovaná. Souvislé řádky jsou obvykle v souvislých blocích, takže se hlavy posouvají málo o málo. Dokonce když není důvod, groupování souvislých bloků probíhá souvisle, takže nejsou potřeba dlouhé posuny.

Dokonce reakční doba může být při sekvenčním čtení nižší. To závisí na HW a na souborovém operačním systému. Disk je obvykle nastaven k minimalizaci reakční doby, ale je možné, že disk potřebuje celou otáčku, než je schopen načíst další blok, což drasticky zpomaluje dobu přístupu.

#### 7.3.7 Cena nesekvenčního přístupu

Cena disku je větší, když jsou řádky vyzvedávány v jiném pořadí, než je jejich fyzické uložení. Normální tabulky jsou mnohem větší, než vyrovnávací paměť, takže jen malá část tabulky může být v paměti. Když je tabulka čtena zpřeházeně, jen málo řádků najdeme ve vyrovnávací paměti. Obvykle je pro každý řádek čten blok.

Když čteme zpřeházeně, je zde obvykle velké časové zdržení i rotační prodleva. Zkrátka při nesequenčním čtení je přístup mnohem pomalejší než při sekvenčním.

### 7.3.8 Cena přístupu podle ID

Nejjednodušší forma nesequenčního přístupu je přístup pomocí rowid. Rowid specifikuje fyzické uložení řádku a jeho blok, takže jeho cena je podobná jako při sekvenčním.

### 7.3.9 Cena indexového přístupu

Zde je přidána cena vážící se k nalezení řádky přes index. Index sám je uložen na disku a musí být načten do paměti.

Engin užívající index má dvě možnosti. Jedna je vyzvednout řádek podle klíčové hodnoty. To vypadá asi takto:

```
SELECT company,order_num
FROM customer,orders
WHERE customer.customer_num=orders.customer_num
```

Jedna tabulka, pravděpodobně customer, bude čtena sekvenčně. Její hodnota customer\_num je použita k prohledání indexu na sloupec customer\_num v tabulce orders. Když matchuje, je řádek přečten.

Náhled do indexu pracuje sestupně od kořenového uzlu k listům. Kořen je obvykle (protože je často používán) umístěn v paměti. Přítomnost náhodného koncového uzlu v paměti závisí na velikosti indexu a ukazuje se nepravděpodobnou. Když je tabulka hodně velká, je počet listových bloků o mnoho větší než vyrovnávací paměť, skoro každé hledání koncového uzlu musí být načteno znovu. Počet načítaných bloků je přibližně  $2r$ , kde  $r$  je počet hledaných řádků. Ačkoliv je to drahé, oproti sekvenčnímu  $r^2$  je to zlepšení.

Engin může také číst index sekvenčně ( $r+i$ ,  $i$  počet listů).

### 7.3.10 Nízká cena malých tabulek

Každá tabulka, která se vejde do vyrovnávací paměti je malá (cca 4 bloky a méně).

### 7.3.11 Cena síťového přístupu

Přesun dat přes síť je, co se týče prodlev, ošidný.

Rozeznáváme tu dva přístupy

- přes INFORMIX-NET aplikace posílající dotazy jinému stroji.
- přes INFORMIX-STAR (distribuovaná data), rys INFORMIX-ONLINE engin může číst řádky databáze z jiného stroje.

## 7.4 Rychlé dotazy

Obecně

- čtou málo řádků
- vyhýbají se třídění, třídí málo řádků, nebo třídí podle jednoduchého klíče
- čtou data sekvenčně

Jak toho dosáhnout není vždy zřejmé. Specifické metody závisí velmi silně na detailech aplikace a databáze. Následující část navrhuje obecná doporučení.

### 7.4.1 Příprava testovacího prostředí

Prvně vyberte jeden dotaz, který je příliš pomalý. Potom nastavte prostředí, v kterém ho chcete odhadovat, schopné opakovat dotaz. Bez tohoto prostředí si nikdy nebudete jisti, zda změna pomohla či nikoliv.

Když užíváte víceuživatelský systém nebo síť, kde se podmínky mění v závislosti na denní době, je třeba provádět experimenty ve stejnou dobu. Můžete třeba vždy pracovat v noci.

Pokud je dotaz součástí složitějšího programu, vytáhněte z něj `SELECT` a spouštějte ho interaktivně, nebo ho zabudujte do jednoduchého programu.

Skutečným dotazům trvá dlouho, než se provedou, tak si můžete připravit menší model DB, v které můžete dotazy testovat rychleji. Může to pomoci, ale musíte si být vědomi následujících problémů:

- optimizér se může rozhodovat v malé db jinak než ve velké, i když relativní velikosti tabulek jsou stejné. Ověřte, zda dotazovací plány jsou v obou případech stejné
- Doba vykonávání je zřídka lineární funkcí velikosti tabulky.

Proto každé vaše rozhodnutí na malém modelu musí být prozatímni dokud si ho neověříte na velké.

### 7.4.2 Studium schématu

Studium definic všech tabulek, pohledů a indexů užitých v DB. Můžete prověřit detaily interaktivně užitím `Table option` INFORMIXu-SQL. Dejte dobrý pozor na indexy, datové typy sloupců užitých v joinovacích podmínkách, na řazení, na pohledy. Čím lépe porozumíte tabulkám, tím lépe porozumíte dotazovacímu plánu vybranému optimizérem.

### 7.4.3 Studium Dotazového plánu

Užijte SET EXPLAIN ON k rozluštění užitého dotazového plánu. Zde je pár motivů:

- joinují se neočekávané tabulky.
- zmínky o dočasných souborech značí, že výstup byl napsán do dočasné tabulky a pak setříděn.
- užití sekvenčního přístupu pro druhou nebo následující tabulku v joinu značí, že tabulka bude přečtená celá pro každý vybraný řádek každé předchozí tabulky.
- autoindex znamená, že se engin bude zdržovat konstrukcí indexu, aby se vyhnul vícenásobnému sekvenčnímu přístupu
- užití sekvenčního přístupu při procházení první tabulky může být marnotratné, když potřebujeme pouze málo řádků

### 7.4.4 Modifikace dotazu

Nyní, když rozumíte, co dotaz dělá, se koukneme na způsoby jak vyhodnocování zlehčit. Následující návrhy odpovídají předchozímu seznamu.

### 7.4.5 Přepsání joinů prostřednictvím pohledů

Můžete přepsat joinovací dotaz na joinovací pohled joinových pohledů. Když přepíšete dotaz přímo na joinování méně tabulek, můžete dostávat jednodušší plán dotazu.

### 7.4.6 Vyhnout se nebo zjednodušit třídění

Třídění není nezbytně špatná věc. Enginovy třídící algoritmy jsou vyladěny k extrémní výkonnosti. Jsou určité tak rychlé jako některá externí třídění, která můžete aplikovat na stejná data. Pokud je třídění děláno příležitostně, nebo na málo výstupních řádcích, není potřeba se mu vyhýbat.

Ale měli by jste se vyvarovat jednoduchému opakovanému třídění velkých tabulek. Optimizér se vyhýbá třídění, jakmile může produkovat výstup ve správném pořadí automaticky užitím indexu.

Zde je několik faktorů, která zabraňují optimizéru užít index:

- Jeden nebo více řadících sloupců není zahrnuto v indexu
- Sloupce jsou vyjmenovány v jiném pořadí v indexu a v ORDER BY nebo GROUP klauzuli
- Řadící sloupce jsou brány z různých tabulek

#### 7.4.7 Odstranění sekvenčního přístupu do velkých tabulek

Sekvenční přístup

Sekvenční přístup do tabulky vypadá na první pohled zlověstně, protože hrozí číst každý řádek tabulky pro všechny předchozí. Měli by jste být schopni rozhodnout, kolikrát to bude možná jen párkrát, ale možná stokrát či tisíckrát. Když je tabulka malá, nevadí číst ji znovu a znovu. Tabulka bude celá v paměti. Sekvenční čtení v paměti může být rychlejší než čtení stejné tabulky přes index, zvlášť když indexové stránky odstraní z paměti jiné používané stránky.

Když je tabulka větší než pár stránek, opakované sekvenční čtení je pro vykonávání smrtelné. Jedna cesta, jak tomu předcházet je nabídnout index na joinovací sloupec.

Index zabírá místo úměrně k šíři klíčových hodnot a počtu řádků. Tedy engin musí modifikovat index kdykoliv jsou řádky vkládány, mazány, nebo modifikovány. To zpomaluje tyto operace. Když je to nezbytné, můžete užít DROP INDEX k odstranění indexu po sérii dotazů, tak uvolnit místo a usnadnit modifikaci.

#### 7.4.8 Užití sjednocení k vyhnutí se sekvenčnímu přístupu

Jisté formy WHERE klauzule přinucují optimizér k sekvenčnímu přístupu, i když index na testovaném sloupci existuje. Následující dotaz si vynucuje sekvenční přístup do tabulky orders:

```
SELECT * FROM orders
WHERE (customer_num=104 AND order_num>1001) OR order_num=1008
```

Klíčový moment je, že jsou vybírány dvě (nebo více) oddělené množiny řádků. Množina je definována pomocí relačních výrazů, které jsou spojeny pomocí OR. V příkladu je jedna množina určena (customer\_num=104 AND order\_num>1001) a druhá order\_num=1008.

Optimizér používá sekvenčního přístupu, i když jsou sloupce customer\_num a order\_num indexované.

Tento dotaz můžete urychlit úpravou na UNION dotaz. Napište oddělené SELECTy pro každou množinu řádků a spojte je pomocí UNION. Zde je přepsaný předchozí případ:

```
SELECT * FROM orders
WHERE (customer_num=104 AND order_num>1001)

UNION

SELECT * FROM orders WHERE order_num=1008
```

Optimizér užije index pro oba dotazy.

#### 7.4.9 Nahrazení autoindexů indexy

Když dotazový plán zahrnuje autoindexy na velké tabulky, berte to jako doporučení optimizéru, že by na tom sloupci měl být index. Je možné nechat engine vystavět a zrušit index, když provádíte dotaz pouze příležitostně, ale když dotaz kladete často, ušetříte čas uděláním řádného dotazu.

#### 7.4.10 Užití smíšeného indexu

Optimizér může užít smíšený index (co pokrývá více sloupců) různými způsoby. Například unikátnost abc. Indexy na sloupce a,b,c může užít následujícím způsobem:

Vyhodnotí filtr na a, zjoinuje a k jiné tabulce, použije ORDER BY na sloupce a, ab, nebo abc

Když vaše aplikace provádí některé dlouhé dotazy a všechny třídí podle stejných sloupců, můžete ušetřit čas vytvořením složeného indexu na tyto sloupce. V důsledku vykonáte třídění jednou a uložíte výstup pro užití v každém dotazu.

#### 7.4.11 Užití bcheck na podezřelé indexy

Na některých enginech je možné, že je užití indexu neefektivní, protože je index poškozen. Když je dotaz užívající index výslovně pomalý, užíjte utilitu bcheck k otestování integrity a opravení indexu (když je to nezbytné). (utilita tbcheck je to samé pro INFORMIX-ONLINE)

#### 7.4.12 Zrušení a přestavění indexu po UPDATE

Po velkých updatech (po přemístění čtvrtiny nebo více řádků tabulky) se struktura indexu může stát neefektivní. Když se index zdá být méně efektivní, než by měl být a přesto bcheck nehlásí žádné chyby, zkuste ho zrušit a znovu vytvořit.

#### 7.4.13 Vyhnout se vztažným poddotazům

Korelované poddotazy jsou takové, které zahrnují vybraný seznam sloupců hlavního dotazu ve WHERE klauzuli poddotazu. Protože se odpovědi poddotazu mohou pro každý řádek lišit, je třeba poddotaz provádět pro každý řádek. Toto může být velmi časově náročné. Naneštěstí bez vztažných poddotazů některé věci v SQL nejdou.

Když uvidíte poddotazy v časově náročném SELECTu, koukněte se, jestli je vztažný. (Vztažný není, když žádné hodnoty sloupců z hlavního dotazu nejsou porovnávány s poddotazem-to je prováděn pouze jednou), zkuste ho přepsat a vyhnout se vztahu. Když nemůžete, zkuste redukovat počet testovaných řádků, například zkuste přidat jiný výraz do WHERE, nebo zkuste vybrat patřičné řádky do dočasné tabulky a hledat pouze v ní.

#### 7.4.14 Vyhnout se obtížným regulárním výrazům

MATCHES a LIKE podporují wildcards, zámě jako regulární výrazy. Některé regulární výrazy jsou pro engin více obtížné než jiné. Wildcarda na počáteční pozici jako v následujícím příkladu (najdi zákazníky, jejichž jméno nekončí na y), nutí engin vyzkoušet každou hodnotu sloupce:

```
SELECT * FROM customer WHERE fname NOT LIKE "%y"
```

Optimizér nemůže použít index pro takový filtr, i kdyby nějaký měl. Toto ho nutí k sekvencnímu postupu tabulkou. Když je tento dotaz druhý nebo později v joinu, dotaz je pomalý.

Když je obtížné testování regulárních výrazů nezbytné, vyhněte se kombinaci s joinem. Nejdříve zpracujte jednu tabulku pomocí regulárního výrazu a vyberte vysněné řádky. Uložte výsledek do dočasné tabulky a připojte k jiným.

Regulární výrazy s wildcardmi jenom uprostřed a nakonci porovnávaných nevyklučuje použití indexu, pokud existuje, ale stejně se jedná o dost pomalou operaci. V závislostech na datech ve sloupci, můžete upravit některé výrazy na porovnávání. Např:

```
SELECT * FROM customer WHERE zipcode LIKE "98_ _ _"
```

Můžete nahradit

```
SELECT *  
FROM customer  
WHERE (zipcode > "98000") AND (zipcode < 99000)
```

#### 7.4.15 Vyhnout se nevýchozím podřetězcům

Filtry založené na nevýchozích podřetězcích sloupců také vyžadují, aby byla každá hodnota testovaná př:

```
SELECT * FROM customer WHERE zipcode[4,5] > "50"
```

Optimizér neužije index i když existuje.

Optimizér užívá index pouze na výchozí hodnoty sloupce.

Takovýto dotaz může být často přepsán jako BETWEEN test na celý sloupec, a může být rychlejší.

#### 7.4.16 Užití dočasných tabulek k urychlení dotazů

Vystavení dočasné setříděné podmnožiny tabulky může někdy urychlit dotaz. Může vám pomoci vyhnout se vícenásobnému třídění a může zjednodušit práci optimizéru.

## 8 Tvorba a užití uložených procedur

### 8.1 Přehled

Následující referát se pokusí ozřejmit způsob tvorby tzv. uložených procedur (Stored Procedures), které se skládají z příkazů SQL a SPL (Stored Procedure Language), a jejich uložení do databáze.

Uložená procedura je vzhledem k SQL uživatelskou funkcí, která je ve spustitelném formátu uložena jako jeden z objektů databáze. Příkazy jazyka SPL se mohou vyskytovat pouze uvnitř konstruktů CREATE PROCEDURE a CREATE PROCEDURE FROM. Tyto příkazy je možné využít v prostředí některých SQL API (jako např. INFORMIX-ESQL/C a INFORMIX-ESQL/COBOL).

Uložení SP ve spustitelném formátu přináší výhodu v případě, že procedura obsahuje nějakou často volanou část kódu, u které odpadá opakované a časově náročné kompilování a také při požadavku na skrytí příkazů tvořících tělo procedury před uživatelem využívajícím danou funkčnost procedury.

Jelikož je SP objektem DB, je přístupná každé aktivní databázové aplikaci, z čehož plyne, že jednu SP může využívat více běžících aplikací.

Uživatel má dále samozřejmě možnost ovlivnit práva pro spuštění a přístup ke každé své SP.

Volání SP je možno využít také k získání hodnot při volání příkazů SQL.

### 8.2 Tvorba uložených procedur

K vytvoření SP je nutné uzavřít příslušné SQL příkazy do konstruktů CREATE PROCEDURE. K ovlivnění běhu operací uvnitř procedury slouží příkazy SPL, které zahrnují například IF, FOR a další, zmíněné níže.

#### 8.2.1 Metoda DB-Access

Při této metodě je SP tvořena konstruktem CREATE PROCEDURE a všemi příkazy uvnitř těla funkce (viz. následující příklad)

```
CREATE PROCEDURE getuid (name CHAR(10)) - začátek definice
RETURNING INT;
DEFINE uid INT;
SELECT uid FROM db WHERE name = cname;
RETURN uid;
END PROCEDURE
DOCUMENT 'Tato procedura vrati UID pro dane jmeno'
WITH LISTING IN tmp/list; - konec definice
```

Průběh překladu bude díky poslední řádce uložen v souboru tmp/list v domovském adresáři uživatele



## 8.3 Využití SQL API

Tato metoda vyžaduje uložení definice SP do zvoleného souboru, který do kódu zakompilujeme pomocí CREATE PROCEDURE FROM. Např. pokud by tedy výše uvedený příklad procedury getuid byl uložen v souboru getuid\_source, pak by příkaz:

```
CREATE PROCEDURE FROM getuid_source;
```

zavedl do databáze funkci getuid a program, využívající tuto funkci, by mohl vypadat následovně:

```
#include <stdio.h>
#include sqlca;
#include sqllda;
main() {
    $database db
    $create procedure from 'getuid_source'
}
```

### 8.3.1 Komentář a dokumentace SP

Jak již naznačil příklad procedury getuid, jsou všechny znaky následující po '-' považovány za komentář. Tento znak se může vyskytovat kdekoli na řádce.

Řetězec následující po příkazu DOCUMENT, jakožto i celá procedura i se zkompilevaným kódem, je zanesena do systémové tabulky sysprocbody, ze které je možné ho z informačních důvodů také získat.

## 8.4 Diagnóza překladových chyb

Pokud překlad procedury z nějakého důvodu selže, je nutné zjistit příčinu a místo kde k chybě došlo. Způsob zjištění problému se liší podle metody přístupu k databázi.

### 8.4.1 Metoda DB-Access

Pokud po neúspěšném pokusu o zkompilevání SP uživatel zvolí v menu SQL volbu Modify, bude se kurzor nacházet právě na místě, kde se chyba vyskytla.

### 8.4.2 Přístup pomocí SQL-API

Pokud selže příkaz CREATE PROCEDURE FROM, server nastaví proměnnou SQLCODE objektu SQLCA na negativní hodnotu a pátý prvek pole SQLERRD naplní offsetem chybného znaku v souboru.

Tabulka: Přístup k chybovým kódům v různých verzích SQL API

ESQL/C sqlca.sqlcode / SQLCODE  sqlca.sqlerrd[4]	ESQL/FORTRAN sqlcod  sqlca.sqlerr(5)	ESQL/COBOL SQLCODE OF SQLCA SQLERRD[5] OF SQLCA
---	---	---

### 8.4.3 Způsob uložení procedury v databázi

Jak již bylo zmíněno výše, každý řádek systémové tabulky sysprocbody odpovídá jedné SP. Obsahuje sloupce procid, data, datakey, kde datakey určuje druh dat nesených sloupcem data a procid jednoznačně určuje SP v systémové tabulce sysprocedures. Pokud sloupec datakey obsahuje znak D, znamená to, že sloupec data nese řádek z dokumentace SP, pokud datakey obsahuje T, značí to, že se jedná o řádek z těla procedury. Zmíněnou vlastnost je možno využít např. tak jako v následujícím příkladě:

```
SELECT data FROM informix.sysprocbody
WHERE datakey = 'T'
AND procid = ( SELECT procid FROM informix.sysprocedures
WHERE informix.sysprocedures.procname = 'getuid' )
```

nebo

```
SELECT data FROM informix.sysprocbody
WHERE datakey = 'D'
AND procid = ( SELECT procid FROM informix.sysprocedures
WHERE informix.sysprocedures.procname = 'getuid')
```

kde budou výsledkem řádky obsahující tělo a v druhém pak řádky dokumentace SP getuid.

## 8.5 Spuštění procedury

Proceduru je možné vyvolat několika způsoby, prvním je volání jako součást příkazů LET

```
CREATE PROCEDURE ...
...
DEFINE userid INT;
...
LET userid = getuid ( 'Nobody' );
...
END PROCEDURE
```

nebo CALL,

```

CREATE PROCEDURE ...
...
DEFINE userid INT;
...
CALL getuid ( 'Nobody' ) RETURNING userid;
...
END PROCEDURE

druhým pak přímé volání EXECUTE PROCEDURE.
EXECUTE PROCEDURE getuid ( 'Nobody' ) INTO userid;

```

## 8.6 Odlaďování procedur

K těmto účelům slouží příkaz TRACE, který umožňuje sledovat tyto objekty:

- proměnné
- argumenty procedur
- návratové hodnoty
- chybové kódy SQL a ISAM

Příkazem

**SET DEBUG FILE** — nastavíme soubor, do kterého se nám bude sledovaný výstup generovat.

**TRACE ON** — nastaví sledování všech příkazů mimo SQL. Obsahy proměnných jsou vytištěny před jejich použitím. Volání jiných procedur a návratové hodnoty jsou také sledovány.

**TRACE PROCEDURE** — nastaví sledování volání procedur a jejich návratových hodnot.

**TRACE** expression — výraz je nejprve vyhodnocen a poté zapsán do výstupního souboru.

### 8.6.1 Znovu zavedení procedury do DB

Jelikož v databázi není možný výskyt dvou procedur se shodným názvem, je nutné před vložením opravené procedury zavolat DROP PROCEDURE pro odstranění stávající verze z databáze a až poté volat CREATE PROCEDURE.

## 8.7 Přístupová práva a SP

Databáze může nést dva typy SP: Typ s právy DBA a typ s právy vlastníka. Typ SP je specifikován při její tvorbě. Následující oddíly popisují rozdíly mezi oběma typy.

### 8.7.1 Práva potřebná pro vytvoření SP

Typ SP s právy DBA mohou vytvářet pouze uživatelé s tímto právem. SP s právy vlastníka může vytvořit kterýkoli uživatel s alespoň právy Resource.

### 8.7.2 Práva potřebná pro spuštění SP

Ke spuštění SP je nutné vlastnit prováděcí nebo DBA právo. Implicitně poskytovaná práva jsou určená, podle toho jestli je SP typu DBA nebo práv vlastníka (a také jestli splňuje normu ANSI) a zbytek práv je určen databázovým serverem.

Pokud je SP typu práva vlastníka, pak je garantováno spouštěcí právo jako PUBLIC. Pokud DB splňuje ANSI, pak server garantuje spouštěcí právo jen vlastníkově a uživatelům s právem DBA.

U SP s právy DBA jsou garantována spouštěcí práva všem uživatelům s právem DBA

### 8.7.3 Typ SP s právy vlastníka

Při spuštění tohoto typu SP server kontroluje existenci a přístupová práva každého odkazovaného objektu. Pro správný běh SP je nutné vlastnit práva na všechny operace prováděné na příslušném objektu. Pokud má vlastník na některých objektech volbu WITH GRANT, pak pokud jinému uživateli garantuje práva pro spuštění (GRANT EXECUTE), pak tento uživatel automaticky získává i tyto práva. Objekty vzniklé za běhu SP patří ovšem vlastníkově SP a ne uživateli, který SP spustil.

### 8.7.4 Typ s právem DBA

Po dobu běhu uživatel nabývá práv DBA. Objekty vzniklé za běhu SP jsou vlastněny aktuálním uživatelem, pokud to ovšem není specifikováno jinak ve zdrojovém kódu SP.

Právo DBA se nedědí vhnížděným voláním jiné procedury, což např. znamená, že když SP s DBA zavolá SP s právy vlastníka, pak tato SP není prováděna s právem DBA.

## 8.8 Proměnné a výrazy

Proměnná není objektem databáze. Je uložena v paměti, z čehož např. plyne, že rollback transakce neobnovuje hodnoty procedurálních proměnných. Před použitím je nutné proměnnou definovat (příkazem DEFINE) a to buď jako globální nebo lokální, poté může být použita na kterémkoli místě v SP. Globálním proměnným musí být přiřazena implicitní hodnota v době kompilace. Typ proměnné může být stejný jako typ kteréhokoli sloupce tabulky mimo typu SERIAL. Typy TEXT a BYTE se definují následovně:

```
DEFINE <var> REFERENCES TEXT/BYTE;
```

Důvodem pro rozdílnou definici je to, že v případě typů TEXT a BYTE proměnná hodnotu přímo neobsahuje, ale pouze se na ni odkazuje.

### 8.8.1 Podřetězce proměnných

U datových typů VAR/N/NV/CHAR, BYTE a TEXT je možné použít následující konstrukci:

```
DEFINE jméno CHAR(15);  
LET jméno[4,7] = 'Pepa';  
SELECT kjméno[1,3] INTO jméno[1,3] FROM zákazník  
WHERE příjmení = 'Soda';
```

Názvy proměnných: Při volbě názvu proměnné nebo procedury mohou nastat konflikty, které server řeší následujícím způsobem:

### 8.8.2 Nejvyšší prioritu mají definované proměnné

- Procedury definované příkazem DEFINE mají přednost před funkcemi SQL
- Funkce SQL mají přednost před procedurami, které nejsou definovány za pomoci DEFINE
- Dále také není možné, aby proměnná nesla jméno kterékoli agregační funkce (např. min, count nebo max). V případě, že se jméno proměnné shoduje s názvem některého ze sloupců tabulky, pak dá DB server vždy přednost proměnné. V odkazu na sloupec tabulky je v tomto případě nutné specifikovat zdrojovou tabulku. Pokud se název proměnné shoduje s názvem některé SQL funkce, pak tuto funkci není možné v rozsahu platnosti zmíněné proměnné použít.

### 8.8.3 Přiřazení hodnot proměnným

K tomuto účelu slouží následující příkazy:

```
LET  
SELECT... INTO  
CALL... RETURNING  
EXECUTE PROCEDURE... INTO
```

## 8.9 Řízení běhu programu

K ovlivnění běhu programu uvnitř SP slouží několik příkazů popsanych v následujících oddílech.

### 8.9.1 Větvení

Klasický konstrukt IF...THEN...ELSE/ELIF...END IF.

### 8.9.2 Cykly

Příkazy pro opakovaný běh některé části programu jsou

**FOR** — pouze konečný počet iterací

**FOREACH** — umožňuje zpracovávat víceřádkové výstupy

**WHILE** — počet iterací nemusí být konečný

Běh programu uvnitř těla cyklu může být rovněž ovlivněn následovně:

**CONTINUE** — pokračuje v následující iteraci cyklu

**EXIT** — opustí cyklus a pokračuje následujícím příkazem

**RETURN** — opustí proceduru s návratovou hodnotou

**RAISE EXCEPTION** — umožňuje např. vyskočit ze zahnízđených cyklů apod.

## 8.10 Zpracování funkcí

### 8.10.1 Funkce OS

Ke spuštění příkazu operačního systému slouží funkce **SYSTEM** jejíž argument je předán systémovému shellu pro zpracování.

### 8.10.2 Rekurze SP

Nejsou kladeny žádné omezení pro rekurzivní volání procedur.

### 8.10.3 Víceřádkové návratové hodnoty

Pokud je pro vrácení hodnoty **SP** použit konstrukt **RETURN...WITH RESUME**, pak nedojde ke ukončení **SP**, ale běh programu pokračuje bezprostředně za tímto příkazem. Ke zpracování návratové hodnoty je vhodné použít příkaz **FOREACH**.

### 8.11 Zpracování výjimek

Konstrukt `ON EXCEPTION`, vyskytující se kdekoli v kódu, umožňuje ošetřit kteroukoli chybu, která může nastat. Na počátku tohoto příkazového bloku specifikujeme, které výjimky má handler zpracovávat, následně ošetříme vzniklé chyby a na konci bloku specifikujeme, jestli se má po ošetření chyby SP ukončit nebo pokračovat v běhu programu.

```
BEGIN
...
ON EXCEPTION IN { -206, -207 } SET err_num
...
END EXCEPTION WITH RESUME
...
END
```

## 9 INFORMIX SQL Triggers

**SQL trigger** (česky třeba „trigr“) je mechanismus umožňující automatické provedení určité množiny SQL operací při výskytu dané události v databázové tabulce. Trigry lze použít pod servery INFORMIX-OnLine, INFORMIX-SE i jejich síťových rozšířeních a to buď v programu DB-Access nebo v některém z vložených jazyků (např. ESQL/C).

### 9.1 Úvod

Trigr se skládá ze dvou částí — *události* a *akce*, která se vykoná při výskytu této události. Událostí může být přidání, smazání nebo přepsání položky v tabulce; akcí může být mimo uvedených příkazů navíc ještě spuštění procedury. Trigry jsou uloženy jako databázové objekty a jsou dostupné v systémových tabulkách *sysstriggers* a *sysstrigbody*. To umožňuje například odstranění redundantních operací z databázových aplikací. Mezi nejdůležitější použití patří udržování referenční integrity, implementace různých pravidel či automatické odvození dalších dat.

### 9.2 Vytvoření trigru

**SQL syntax:**

```
CREATE TRIGGER jméno_trigru Událost Akce
```

Událost je právě jeden z výrazů:

```
INSERT ON tabulka [ REFERENCING NEW AS jméno ]
DELETE ON tabulka [ REFERENCING OLD AS jméno ]
UPDATE OF sloupec,... ON tabulka
    [ REFERENCING [ NEW AS jméno1 ] [ OLD AS jméno2 ] ]
```

a určuje při jaké operaci, na jaké tabulce (u UPDATE také na kterých sloupcích) budou provedeny následující Akce. Příkaz *REFERENCING* umožňuje definovat nové jméno pro tabulku před (*OLD*) nebo po (*NEW*) provedení této operace. Tyto jména lze použít pouze v akci *FOR EACH ROW*.

Akce mohou být (v tomto pořadí):

```
BEFORE Podmíněné příkazy
FOR EACH ROW Podmíněné příkazy
AFTER Podmíněné příkazy
```

a určují kdy a jaké příkazy se budou provádět (viz. Vyvolání trigru). Akce jsou nepovinné pouze v případě, že bylo použito *REFERENCING*, tak je povinná akce *FOR EACH ROW*.

Podmíněné příkazy jsou čárkou oddělené příkazy tvaru:



[ WHEN (Podmínka) ] (Příkaz), ...

Nepovinná Podmínka je běžná SQL podmínka a Příkaz je buď *INSERT*, *UPDATE*, *DELETE* nebo *EXECUTE PROCEDURE*. Příkaz je proveden pouze, pokud je příslušná podmínka splněna, nebo vždy, když není uvedena.

Příkaz *EXECUTE PROCEDURE*, který spouští uloženou proceduru (anglicky „stored procedure“), umožňuje v akci trigrů *FOR EACH ROW* přiřadit výstup do sloupců tabulky pomocí *INTO sloupec,...* (např. *EXECUTE PROCEDURE sum(x,y) INTO xplusy, total*).

### 9.3 Zrušení trigrů

SQL syntax:

```
DROP TRIGGER jméno_trigrů
```

### 9.4 Vyvolání trigrů

Akce trigrů se začínou provádět při vzniku příslušné události, kterou vytvoří nějaký „spouštěcí“ SQL příkaz, v případě, že má provést danou operaci na dané tabulce (popř. i sloupcích). Nejdříve se provede akce *BEFORE*, dále se pro každý řádek, kterého se spouštěcí příkaz týká, vykoná akce *FOR EACH ROW*, poté se provede samotný spouštěcí příkaz a nakonec se vykoná akce *AFTER*.

### 9.5 Omezení v trigrech

Trigry a jejich akce musí splňovat různá omezení, zde je uvedeno několik z nich:

- Vytvoření ani zrušení trigrů nelze provést v uložené proceduře, která byla zavolána během manipulace s daty příslušné tabulky (např. *INSERT INTO tabulka EXECUTE PROCEDURE proc(x)*).
- Pro každou tabulku může být pouze jediný trigr typu *INSERT* a *DELETE*; více trigrů *UPDATE* je přípustných jen, pokud jsou definovány na různých sloupcích. V případě, že spouštěcí příkaz vyvolá více trigrů, jsou jejich akce spojeny do bloků podle typu, kde jsou seřazeny vzestupně podle čísla sloupce (viz. Příklad 2).
- Akce trigrů smí odkazovat na vlastní tabulku pouze v příkazu *SELECT* nebo *UPDATE*, pokud mění jiné sloupce než, kterých se trigr týká.
- Akce jednoho trigrů může způsobit vyvolání jiného trigrů — takto lze trigry „kaskádovat“ maximálně do hloubky 61.

## 9.6 Systémové tabulky o trigrech

Obecné informace o trigrech jsou v tabulce systriggers, která má následující položky:

trigid	SERIAL	identifikační číslo
trigname	CHAR(18)	jméno
owner	CHAR(8)	vlastník
tabid	INT	identifikační číslo tabulky
event	CHAR	událost (I insert, U update, D delete)
old	CHAR(18)	jméno hodnoty před změnou
new	CHAR(18)	jméno hodnoty po změně
mode	CHAR	(rezervováno)

Druhá tabulka, systribody, obsahuje těla trigrů v textové a linearizované formě:

trigid	INT	identifikační číslo
datakey	CHAR	typ D text pro hlavičku A text pro tělo H linearizovaný kód pro hlavičku S linearizovaný kód pro tabulku symbolů B linearizovaný kód tělo
seqno	INT	sekvenční číslo
data	CHAR(256)	text nebo linearizovaný kód

## 9.7 Příklady

### 9.7.1 Příklad 1

```
CREATE TABLE polozky (jmeno CHAR(10), hodnota MONEY(4));

CREATE PROCEDURE kontrola(hodnota MONEY(4))
  IF hodnota <= 0 THEN
    RAISE EXCEPTION -746, 0, "polozka je bezcenna!";
  END IF
END PROCEDURE

-- dovolime vkladat pouze hodnotne polozky
CREATE TRIGGER polozky_trig
  INSERT ON polozky REFERENCING NEW AS post
  FOR EACH ROW (EXECUTE PROCEDURE kontrola(post.hodnota));

-- hmm...
INSERT INTO polozky VALUES("rubl", 0);
```

### 9.7.2 Příklad 2

```
--
-- Vyvolani vice trigru na jedine tabulce
-- (akce nejsou dopsany!)
--
CREATE TABLE tab (a INT, b INT, c INT, d INT);

CREATE TRIGGER trig1 UPDATE OF a, c ON tab
  BEFORE ...
  FOR EACH ROW ...
  AFTER ...

CREATE TRIGGER trig2 UPDATE OF b, d ON tab
  BEFORE ...
  FOR EACH ROW ...
  AFTER ...

-- Nasledujici spousteci prikaz vyvola akce v~tomto poradi:
-- 1. BEFORE trig1
-- 2. BEFORE trig2
-- 3. FOR EACH ROW trig1
-- 4. FOR EACH ROW trig2
-- 5. spousteci prikaz
-- 6. AFTER trig1
-- 7. AFTER trig2
UPDATE tab SET (b, c) = (b + 1, c + 1);
```

### 9.7.3 Příklad 3

```
CREATE TABLE staty (kod CHAR(2), jmeno CHAR(100));
CREATE TABLE mesta (jmeno CHAR(50), kod_statu CHAR(2));

-- se statem chceme vymazat i jeho mesta
CREATE TRIGGER staty_trig
  DELETE ON staty REFERENCING OLD AS pre
  FOR EACH ROW (DELETE FROM mesta WHERE kod_statu = pre.kod);

-- a takle se vporadame s~nepritelem
DELETE FROM staty WHERE jmeno = "Irak";
```

### 9.7.4 Příklad 4

```
CREATE TABLE akcie (
  vlastnik CHAR(8),
```

```

    firma CHAR(50),
    pocet INT, cena MONEY(4),
    vynos MONEY(4));

CREATE PROCEDURE
    prepocitej(pocet INT, cena MONEY(4)) RETURNING MONEY(4)
    RETURN pocet * cena;
END PROCEDURE

-- pri zmene pocetu nebo ceny se automaticky prepocita vynos
CREATE TRIGGER akcie_trig
    UPDATE OF pocet, cena ON akcie
    FOR EACH ROW
        (EXECUTE PROCEDURE prepocitej(pocet, cena) INTO vynos);

-- takhle prijdeme o~polovinu majetku
UPDATE akcie SET cena = cena / 2 WHERE vlastnik = USER;

```

## 10 Informix ESQL/C

### 10.1 Co to je

Informix ESQL/C je nástroj, který umožňuje psát aplikace v jazyce C a při tom využívat toho, co nabízí databáze Informix. Do zdrojového kódu v jazyce C vkládáte SQL-dotazy a speciální hlavičkové soubory. Na tento kód se potom spustí preprocesor ESQL/C a výsledkem je kód v jazyce C. Ten se potom přeloží běžným kompilátorem a výsledkem je spustitelný soubor. Preprocesor ESQL/C překládá zdrojový text ve dvou krocích:

- První krok funguje podobně jako preprocesor u překladače jazyka C. Můžete pomocí něj do zdrojového textu vložit jiný zdrojový text napsaný v ESQL/C, definovat konstanty a umožňuje podmíněný překlad.
- Druhý krok pracuje jako překladač z ESQL/C do jazyka C.

Preprocesor ESQL/C rozlišuje velká a malá písmena. Všechno za "--" se bere jako komentář.

### 10.2 Výrazy pro preprocesor ESQL/C

Kód určený pro preprocesor ESQL/C musí být uvozen buď slovy EXEC SQL (podle standardu) nebo znakem dolaru (specifická pro ESQL/C od Informixu). Musí být také ukončen středníkem.

Preprocesor preprocesoru (dále jen "PP") ESQL/C umožňuje:

- vkládat do zdrojového kódu ESQL/C jiný zdrojový kód  
Jméno souboru se může zadávat buď s uvozovkami nebo bez uvozovek. Je-li ale specifikováno i s cestou, jsou uvozovky nutné.  
Postup hledání souboru:

1. aktuální adresář
2. adresář \$INFORMIXDIR/incl/esql
3. adresář /usr/include

Do zdrojového textu je automaticky vložen soubor **sqlca.h**, který obsahuje definici struktury, do které se ukládají chybové kódy.

Při překladu je automaticky vložen soubor **sqlca.h**.

Další hlavičkové soubory:

**sqllda.h** ??? - pro dynamické definované proměnných

**sqlstype.h** celočíselné konstanty odpovídající SQL dotazům; používá se při DESCRIBE

**sqltypes.h** definice řetězců odpovídajících jazyku C a datovým typům SQL; použití při DESCRIBE

**varchar.h** definuje makra pro práci s typem VARCHAR

**decimal.h** definuje strukturu, do níž se ukládá hodnota typu DECIMAL

**datetime.h** definuje strukturu, do níž se ukládá hodnota typu DATE-TIME nebo INTERVAL

- definovat (define) a rušit definice (undef) symbolů  
Definuje se stejně, jen s tím omezením, že můžete definovat jen symboly a celočíselné konstanty (ne např. makra nebo řetězce). Nezapomeňte na středník na konci!
- podmíněčný překlad částí kódu  
Podmíněčný překlad je omezen na podmínky typu „ifdef“, „ifndef“. Použitelné výrazy jsou „ifdef“, „ifndef“, „elif“, „else“, „endif“.

### 10.3 Host-proměnné

Host-proměnná je proměnná jazyka C, která bude používá i ve výrazech SQL. Při použití musí být uvozena dvojtečkou (podle standardu ANSI) nebo znakem dolaru.

#### 10.3.1 Deklarace

Deklarace musí být označená jedním z těchto dvou způsobů:

1. První řádek deklarace promenné je uvozen znakem dolaru.
2. Proměnná je v bloku označeném EXEC SQL BEGIN DECLARE SECTION a EXEC SQL END DECLARE SECTION.

Standardu ANSI odpovídá označení přes EXEC SQL ...

Druhý způsob je standardní (norma ANSI).

Při deklaraci se hodnoty mohou inicializovat. Má to ale jedno omezení. Je-li součástí inicializační hodnoty řetězec, nesmí obsahovat dvojtečku ani klíčové slovo ESQL/C.

Syntaktická správnost zápisu inicializačních hodnot není kontrolována PP, ale až kompilátorem céčka.

#### 10.3.2 Rozsah platnosti

Pro rozsah platnosti host-proměnné platí stejná pravidla jako pro obyčejné proměnné v céčku. V případě potřeby je možné platnost host-proměnné omezit na blok ohraničený symboly \${ a \$}. Takto ohraničené bloky je možné do sebe vnořovat až do hloubky 16, přičemž celý zdrojový text je brán jako blok hloubky 1. Ohraničování bloků ve standardu ANSI není.

### 10.3.3 Vztah mezi datovými typy C a SQL

Host-proměnné jsou používány jak v jazyce C, tak i v SQL-výrazech. Vztahy mezi datovými typy jsou v této tabulce:

SQL typ	předdefinovaný datový typ ESQL/C	typ v jazyce C
CHAR(n) CHARACTER(n)	fixchar array[n] nebo string array[n+1]	char array[n+1] nebo char *
BYTE TEXT	loc_t	
DATE		long int
DATETIME	datetime nebo dtm_t	
DECIMAL DEC NUMERIC MONEY	decimal nebo dec_t	
SMALLINT		short int
FLOAT DOUBLE PRECISION		double
INTEGER INT		long int
INTERVAL	interval nebo intrvl_t	
SERIAL		long int
SMALLFLOAT REAL		float
VARCHAR(m,x)	varchar[m] nebo string array[m+1]	char array[m+1]

### 10.3.4 Pole

ESQL/C umí pracovat s poli host-proměnných. Pole ale musí být jedno- nebo dvourozměrné a při deklaraci musí být dána jeho velikost.

### 10.3.5 Struktury

Struktury host-proměnných jsou dovoleny a mohou být vnořované. Při použití v SQL-výrazu se proměnná nadeklarovaná jako struktura nahradí seznamem položek, které obsahuje.

Například:

```
$insert into customer values ($cust_rec);
```

znamená totéž, co

```
$insert into customer
values ($cust_rec.c_no, $cust_rec.fname, $cust_rec.lname)
```

### 10.3.6 Definice typů (typedef)

Definice nového typu pro host-proměnné pomocí typedef je možné, ale není dovoleno použít typedef k nadefinování host-proměnné jako vícerozměrného pole nebo *union*.

### 10.3.7 Hodnota NULL

Interpretace databázové hodnoty NULL se liší podle stroje i podle datového typu a nemusí odpovídat interpretaci žádné Cčkovské proměnné. Proto není radno s proměnnou, která má hodnotu NULL, provádět nějaké aritmetické operace.

Na zjištění, zda nějaká host-proměnná má nebo nemá hodnotu NULL, existují funkce **risnull** a **rsetnull**.

### 10.3.8 Deklarace host-proměnných typu ukazatel na znak

Host-proměnnou typu (char \*) je možné nadeklarovat, ale měla by se používat jen pro využití jejího obsahu ve výrazu SQL (neměla by se SQL-dotazem měnit její hodnota).

### 10.3.9 Deklarace host-proměnných jako parametry funkce

Je nutno použít klíčové slovo **parameter**:

```
spocitej (s, id, size)
$parameter char s[20];
$parameter int id;
$int s_size;
select fname into $s from customer
where customer_num = $id;
...
```

Nadeklarování obyčejné proměnné s klíčovým slovem **parameter** vede k nepředvídatelnému chování.

## 10.4 Indikační proměnné (indicator variables)

Indikační proměnné slouží k indikaci toho, že

- do host-proměnné se načetla hodnota NULL
- řetězec načítaný z databáze do host-proměnné je delší, než na kolik je host-proměnná deklarovaná, a byl proto zkrácen

Může být také použita pro vložení hodnoty NULL do databáze.



#### 10.4.1 Deklarace

Indikační proměnná je obyčejná celočíselná proměnná a indikační se stává až při použití s host-proměnnou, jejíž hodnotu chceme testovat. Nadeklarovat ji tedy můžeme například takto:

```
$short nameind;
```

#### 10.4.2 Funkce

- Jestliže se do host-proměnné načetla hodnota NULL, indikační proměnná, asociovaná s touto host-proměnnou, nabyde hodnoty -1. Hodnota host-proměnné může být přitom nesmyslná.
- Po načtení řetězce do host-proměnné indikační proměnná obsahuje délku řetězce před případným zkrácením. To, jestli řetězec byl nebo nebyl zkrácen, poznáte podle struktury sqlca, jejíž obsah se aktualizuje po každém SQL-dotazu.

V případě, že načtená hodnota nebyla NULL ani to nebyl příliš dlouhý řetězec, indikační proměnná obsahuje hodnotu 0.

#### 10.4.3 Použití

Indikační proměnná se stává indikační až tehdy, když je asociovaná s nějakou host-proměnnou. Tato host-proměnná se potom nazývá hlavní proměnná (main). Spojení indikační proměnné s hlavní proměnnou se označuje následovně:

- mezi hlavní proměnnou a indikační proměnnou se vloží dvojtečka nebo znak dolaru. Doporučuje se používat dvojtečku - zápis je přehlednější.
- mezi hlavní proměnnou a indikační proměnnou se vloží klíčové slovo INDICATOR. Toto je standardní způsob.

Příklad:

```
/* Deklarace */
$int cena;
$short cenaind;
/* Užití: */
EXEC SQL select cena into $cena:cenaind;
```

### 10.5 Ošetření chyb

Po každém zavolání nějakého SQL-dotazu kromě DESCRIBE vrací server informace o provedené operaci do struktury **sqlca**.

Struktura sqlca je uložena v souboru **sqlca.h**.

Struktura **sqlca**:

```

struct sqlca_s
{
    long sqlcode;
    char sqlerrm[72];
    char sqlerrp[8];
    long sqlerrd[6];
    struct sqlcaw_s
    {
        char sqlwarn0;
        char sqlwarn1;
        char sqlwarn2;
        char sqlwarn3;
        char sqlwarn4;
        char sqlwarn5;
        char sqlwarn6;
        char sqlwarn7;
    }
}

```

Návratový kód je uložen v položce **sqlcode**.

sqlca.sqlcode < 0	neúspěch
sqlca.sqlcode = 0	úspěch
sqlca.sqlcode = SQLNOTFOUND	vrácena prázdná množina

(SQLNOTFOUND má hodnotu 100)

Aby se nemuselo pořád přistupovat k **sqlca.sqlcode**, je obsah této položky automaticky kopírován do globální proměnné sqlcode.

## 11 Transakce v INFORMIXU

Informix zajišťuje integritu pomocí transakcí.

### 11.1 Co je to transakce?

Transakce je posloupnost databázových operací (SQL příkazů), která má být vykonána buď jako celek, nebo vůbec. Nikdy se nesmí stát, aby byla vykonána pouze její část.

### 11.2 Příkazy používané v souvislosti s transakcemi

**BEGIN WORK** — označuje začátek transakce

**COMMIT WORK** — označuje konec transakce a trvale zapíše všechny změny od počátku transakce

**ROLLBACK WORK** — označuje konec transakce a odvolá všechny změny provedené od začátku transakce

**START DATABASE** — inicializuje nový žurnál a (volitelně) převede databázi na MODE ANSI

**ROLLFORWARD DATABASE** — použije žurnál k obnovení databáze

Příkazy **BEGIN WORK**, **COMMIT WORK**, **ROLLBACK WORK** a **ROLLFORWARD DATABASE** mohou být použity pouze pokud se jedná o databázi s transakcemi. Můžeme použít klauzuli **WITH LOG IN** v příkazech **CREATE DATABASE** nebo **START DATABASE** pro vytvoření nového žurnálu.

### 11.3 Databáze bez transakcí (databáze bez logu)

U databází bez transakcí může dojít k narušení integrity dat chybou uvnitř posloupnosti těsně souvisejících databázových operací. Pokud k takové chybě dojde, je třeba se explicitně postarat o napravení spáchané škody a uvedení databáze do původního stavu.

### 11.4 Databáze s implicitními transakcemi

#### 11.4.1 MODE ANSI

Pokud vytvoříme databázi jako MODE ANSI, a specifikujeme žurnál (log file), pak všechny SQL příkazy jsou automaticky součástí transakce a není třeba používat příkaz **BEGIN WORK**.

#### 11.4.2 non MODE ANSI

V takovéto databázi je nutné explicitně určit začátek transakce příkazem **BEGIN WORK**, předtím, než se začne vykonávat posloupnost příkazů, která se má vykonat jako celek. Tím jsou také všechny změněné záznamy uzamčeny, aby je nemohl nikdo jiný v průběhu transakce měnit. (Ale je možné je prohlížet.)

Pokud nepoužijeme příkaz **BEGIN WORK**, INFORMIX-4GL zachází z každým příkazem, který mění databázi jako s jednoduchou transakcí.

#### 11.5 Omezení při používání transakcí

Protože počet současně uzamčených záznamů je omezen, je dobré se snažit dělat krátké transakce, které mění pouze několik záznamů. Pokud je předpoklad, že počet záznamů, které budou v průběhu transakce změněny, je možné použít zamýkání celé tabulky, dokud není transakce ukončena.

#### 11.6 Používání transakcí

Bez ohledu na to, zda jsou transakce implicitní nebo explicitní, je možné ukončit transakci příkazem **COMMIT WORK** jakmile je možné zaznamenat změny do databáze. Naopak příkazem **ROLLBACK WORK** můžeme vrátit databázi do stavu v jakém byla před započítím transakce (až na výjimky uvedené v následujícím odstavci). Oba tyto příkazy uvolní všechny záznamy a tabulky uzamknuté v průběhu transakce.

#### 11.7 Akce u kterých není možné provést ROLLBACK

- **GRANT** a **REVOKE**
- akce měnící jména a počet tabulek
- změny počtu, jmen, datových typů nebo indexů u sloupců

Tyto akce by neměly být používány v explicitních transakcích.

## 12 Pohledy v INFORMIXU

Pohledy v Informixu umožňují vytvoření entit chovajících se jako tabulka pomocí dotazu nad tabulkou, spojením tabulek i nad jinými pohledy.

### 12.1 Možná použití pohledů

- zpřístupnění jen některých sloupců nebo řádků tabulky
- vkládání a update hodnot splňujících určité podmínky
- přístup k odvozeným datům bez nutnosti jejich redundantního uložení v databázi
- je možné skrýt detaily spojení tabulek

### 12.2 Vytvoření pohledu

```
CREATE VIEW jmeno_pohledu [ ( seznam_sloupcu ) ]  
      AS SELECT-prikaz [ WITH CHECK OPTION ]
```

- *jmeno\_pohledu* je identifikátor
- *seznam\_sloupcu* je seznam identifikátorů pro sloupce vytvářeného pohledu (pokud není uveden, pohled zdědí jména sloupců z tabulek ze kterých byl vytvořen)
- datové typy sloupců pohledu se dědí z tabulek

pozn.:

- uživatel musí mít právo **SELECT** na všechny potřebné tabulky
- v databázích vytvořených jako MODE ANSI se k pohledu přistupuje tečkovou notací přes jméno vlastníka pohledu a je nutné uvést jméno vlastníka při přístupu k 'cizímu' pohledu
- v databázích, které nejsou MODE ANSI není explicitní uvádění vlastníka nutné

### 12.3 Omezení pro pohledy

- pohledy není možné indexovat
- nelze použít **ALTER**, **RENAME** a **RENAME COLUMN**
- V příkazu **SELECT** nelze použít **INTO TEMP**, **UNION** a **ORDER BY**

## 12.4 Chování pohledů

Pohled se chová stejně jako tabulka se jménem *jmeno\_pohledu* a sestává s řádků a sloupců, které vrátil příkaz **SELECT**. Příkaz **SELECT** je vyvolán při každém použití pohledu, takže změny provedené v tabulkách se v něm projevují.

## 12.5 Vliv změn v databázi na pohledy

- při zrušení tabulky či pohledu jsou zrušeny o pohledy z nich vytvořené
- při přejmenování tabulky se pohled naváže na nové jméno
- jestliže byl pohled vytvořen použitím **SELECT \*** obsahuje pouze sloupce existující při jeho vytvoření, ale nikoliv ty, které byly do tabulek později přidány
- při zrušení sloupce tabulky se pohled nezmění, ale dotazy na něj končí chybou

### Modifikace tabulek přes pohledy

1. Nelze modifikovat:

- spojení tabulek
- pohledy ve kterých bylo použito **GROUP BY**, **DISTINCT**, nebo **UNIQUE**

2. Mazání záznamů: lze kdykoliv neplatí bod 1

3. Updatování záznamů: jako bod 1 & žádný sloupec neobsahuje výraz vytvořený z konstatní hodnoty (lze updatovat, pokud zde použijeme hodnoty NULL)

4. Vkládání záznamů: jako update

5. **WITH CHECK OPTION** zajistí, že při modifikaci tabulek přes pohled se uplatní pouze takové změny, které jsou v souladu s definicí pohledu (tj. změněný záznam opět padne do pohledu)

## 12.6 Zrušení pohledu

Ke zrušení pohledu slouží příkaz **DROP VIEW**.

## 13 Práce s BLOB (Binary Large Objects)

Pro demonstraci práce s bloby slouží databáze **stores5**, která se nainstaluje pomocí skriptu **esqldemo5**. Ten je umístěn v adresáři **\$INFORMIXDIR/bin** a při spuštění ve tvaru **esqldemo5 jméno.db** vytvoří databázi, pojmenovanou **jméno\_db**, přidělí vám v ní administrátorská práva a také vám nainstaluje řadu zdrojových textů v **esql**, které slouží k demonstraci práce s bloby.

Příklady využívají tabulku databáze **stores5 catalog**, která kromě klíče typu **long catalog\_num** obsahuje blob typu **TEXT cat\_descr** a blob typu **BYTE cat\_picture**.

### 13.1 Datový typ

Sloupce tabulky, které slouží k uchovávání blobů musí mít typ **TEXT** nebo **BYTE**. S objekty tohoto typu nejde v SQL dotazech dělat žádné testy, jedná se vlastně o čistě uživatelský typ, interpretaci jeho hodnot si musí zajistit uživatelská aplikace.

### 13.2 Programování

Na zápis blobu do databáze nebo fetch z databáze se používá pomocná struktura — **LOKÁTOR**. Neobsahuje sice vlastní data BLOBu ale pointer na ně a mnoho dalších pomocných údajů pro nastavení parametrů operace. Je to tedy typ céčkovské proměnné pro práci v **ESQL/C** se sloupci typu **TEXT** a **BYTE**.

Definice struktury a všech použitelných konstant je v **locator.h**.  
(konvence komentářů: kdo atribut využívá - k čemu)

```
typedef struct {
    short loc_type; // uživatel - určení typu LOKÁTORU
    union {
        //jestliže je LOKÁTOR na hodnotě LOCKMEMORY
        struct {
            long lc_bufsize; //uživatel - velikost bufferu pro blob
            char* lc_buffer; //uživatel - blob v do
                                //uživatelské oblasti
            char* lc_currdata_p; //vnitřní - blob v db oblasti
            int lc_mflags; //vnitřní - paměťová příznaky
        } lc_mem;
        //jestliže je LOKÁTOR na LOCFNAME nebo LOCFILE
        struct {
            char* lc_fname; //uživatel - jméno souboru pro blob
            int lc_mode; //uživatel - nastavení práv při
                            //vytváření souboru
            int lc_fd; //uživatel - file deskriptor
        }
    }
};
```

```

        long lc_position; //vnitřní - prohledávací pozice
    } lc_file;
} lc_union;

long loc_indicator; //uživatel nebo system - indikátor výsledku
long loc_type;      //system - typ blobu
long loc_size;      //uživatel nebo system - velikost
                    //blobu nebo -1
int loc_status;     //system - status po provedení operace
char* loc_user_env; //uživatel - pro vlastní využití
long loc_xfercount; //vnitřní - velikost přenosu

//uživatelské funkce pro otevření, zavření lokátoru,
//čtení a zápis blobu
int (*loc_open) ();
int (*loc_close) ();
int (*loc_read) ();
int (*loc_write) ();

int loc_oflags; //uživatel a vnitřní: příznaky
} loc_t;

Pro zkrácení zápisu při přístupu k jednotlivým položkám jsou definována
makra:

#define loc_fname      lc_union_lc_file.lc_fname
#define loc_fd         lc_union.lc_file.lc_fd
#define loc_position   lc_union.lc_file.lc_position
#define loc_bufsize    lc_union.lc_mem.lc_bufsize
#define loc_buffer      lc_union.lc_mem.lc_buffer
#define loc_currdata_p lc_union.lc_mem.lc_currdata_p
#define loc_mflags     lc_union.lc_mem.lc_mflags

```

### 13.3 Políčka společná všem datovým umístěním

- **loc\_indicátor** -1 identifikuje NULL hodnotu uživatel použije při ukládání NULL do DB, systém při načtení NULL z DB
- **loc\_status** 0 jestliže operace skončila úspěchem, jinak identifikuje chybu
- **loc\_type** – jestli se jedná o TEXT nebo BYTE
- **loc\_oflags** – hodnoty:
  - LOC\_RDONLY, LOC\_WONLY, LOC\_APPEND — přístup souborům
  - LOC\_TEMPFILE — 4GL tempfile blob



- LOC\_USEALL — ekvivalentní nastavení loc.size na -1, aby při čtení blobu (ze souboru) bylo vždy čteno po největších možných kusech (má smysl jen pro čtení ze souborů)
- LOC\_ALLOC — pro loc.open, aby vždy uvolnil a nově alokoval buffer
- LOC\_DESCRIPTOR — má smysl při práci s WORM (write once read many) disky

Lokátor poskytuje prostředky pro čtení/zápis blobů do paměti, do otevřeného diskového souboru udaného descriptor, do souboru udaného jménem nebo při předefinování obslužných funkcí funguje podle vůle uživatele. Způsob použití je určen nastavením atributu loc.type.

### 13.4 BLOB v paměti

tj. nastaveno loc.type == LOCKMEMORY

Význam atributů lokátoru:

- loc.buffer — umístění dat blobu v paměti – tj. odkud při zápisu do db nebo kam při čtení z db jdou data blobu
- loc.bufsize — velikost bufferu nebo -1

Je-li loc.bufsize -1, použije při čtení server malloc a nastaví správnou hodnotu (i v loc.buffer). Navíc jestliže je do loc.buffer alokovaná paměť (např. při opakovaném dotazu), server jí nejprve uvolní.

Jestliže je při čtení jiná velikost dat a loc.bufsize, je loc.status nastaven na zápornou hodnotu a skutečnou velikost dat server oznámí v loc.indicator. Jestliže je při zápisu loc.bufsize menší než loc.size, je vrácena chyba.

- loc.size — velikost blobu nebo -1, při čtení vyplní systém, při zápisu uživatel

#### 13.4.1 Příklad: načtení blobu do paměti

```
$long cat_num;
$loc_t cat_descr

cat_descr.loc_loctype = LOCKMEMORY;
cat_descr.loc_bufsize = -1;
cat_descr.loc_oflags = 0;

$select catalog_num, cat_descr
into $cat_num, $cat_descr from catalog
where catalog_num = $cat_num;
```

### 13.4.2 Příklad: Zapsání blobu z paměti

```
$long cat_num;
$loc_t cat_descr;

char* ans = (char*) malloc(BUFFSZ);

/*zde by melo byt nacteni retezce pro blob do promenne ans */

cat_descr.loc_loctype = LOCKMEMORY;
cat_descr.loc_buffer = ans;
cat_descr.loc_buffsize = BUFFSZ;
cat_descr.loc_size = strlen(ans);

$update catalog set cat_descr = $cat_descr
  where catalog_num = $cat_num;
```

### 13.5 Blob v otevřeném souboru udaném descriptorom

tj. loc.type = LOCFILE

File deskriptor pro otevřený soubor musí být zapsán do loc\_fd, způsob otevření souboru musí být specifikován v loc\_oflags (LOC\_RDONLY, LOC\_WRONLY, LOC\_APPEND). Blob se čte nebo zapisuje od aktuální pozice v souboru. Při čtení blobu ze souboru se čte loc\_size bytů nebo do konce souboru, pokud je loc\_size == -1. Při zápisu blobu do souboru, server zapíše všechna data blobu a pak jejich délku nastaví v loc\_size.

```
$loc_t cat_descr;
$long cat_num;
int fd = open("jméno souboru", O_WRONLY);

cat_descr.loc_loctype = LOCFILE;
cat_descr.loc_fd = fd;
cat_descr.loc_oflags = LOC_APPEND;
$select catalog_num, cat_descr nto $cat_num, $cat_descr
  from catalog where catalog_num = $cat_num;
```

### 13.6 Blob v souboru udaném jménem

tj. nastavení loc\_type na LOCFNAME

Je potřeba nastavit loc\_fname na jméno souboru a loc\_oflags na způsob otevření. Server soubor otevře, nastaví si loc\_fd a pak už je vše jako v předchozím případě.

## 13.7 Blob v uživatelem definované oblasti

tj. nastavení `loc_loctype` na `LOCUSER`

V tomto případě má uživatel plnou kontrolu nad přístupem k blobům v databázi, musí si tedy definovat vlastní funkce pro otevření, zavření, načtení blobu do db a zápis blobu z db a pointery na ně nastavit v patřičných atributech lokátoru. Počet a význam parametrů těchto funkcí je pevně určen. Při provádění SQL příkazu, který vede na načtení nebo zápis blobu, pak server volá tyto uživatelské funkce.

Uživatel může pro vlastní potřebu využít `loc_usr_env`, `loc_xfercount` a `loc_union` (server tyto položky buď sám nepoužívá, nebo by je používaly pouze jeho standardní funkce pro otevření, zápis, čtení a zavření lokátoru).

### 13.7.1 Otevření lokátoru

```
openblob(adloc, oflags)
loc_t * adloc;
int oflags;
```

Funkce na otevření dostane dva parametry — adresu lokátoru a příznak `R_ONLY` nebo `W_ONLY`, který specifikuje zda bude lokátor použit k zápisu do databáze nebo načtení z databáze. Funkce vrátí 0 při úspěchu a -1 při chybě.

### 13.7.2 Zavření lokátoru

```
closeblob(adloc)
loc_t *adloc;
```

### 13.7.3 Načtení lokátoru

```
readblob(adloc, bufp, ntored)
loc_t *adloc;
char* bufp;
int ntored;
```

Argumenty mají význam adresy lokátoru, místa kam se mají zapisovat data z blobu a počet bytů, co se má přečíst. Fce vrátí počet přečtených bytů nebo EOF. Pokud totiž uživatel čte s `loc_size` nastaveným na -1, server volá `read` funkci dokud nevrátí EOF. Pokud funkce vrátí menší hodnotu, než požadovaný počet přečtených byte, předpokládá se EOF.

### 13.7.4 Zápis lokátoru

```
writeblob(adloc, bufp, ntowrite)
loc_t* adloc;
char* bufp;
```

```
int ntored;
```

Argumenty mají význam adresy lokátoru, adresy odkud se mají data číst a počtu bytů, co se má zapsat.

## 14 Fyzický a logický log v INFORMIXu

### 14.1 Fyzický log

**Fyzický log** je vlastně záznamem předobrazů stránek vytvářeným za stavu fyzické i logické konzistence v prostoru *dbspace*. Umožňuje rekonstruovat poslední stav konzistence databáze. Jeho umístění specifikuje parametr *PHYSDBS*, velikost v KB parametr *PHYSFILE*. Je-li systém OnLine v klidovém stavu, je možné fyzický log přesunout z *root dbspace*, kde je umístěn defaultně, do jiného *dbspace*, čímž se může zlepšit celkový výkon.

Fyzický log se skládá z několik po sobě jdoucích stránek uložených na disku. Ty obsahují kopie dat ze stránek systému OnLine s výjimkou stránek z prostoru *blob space*, jež jsou zaznamenávány jiným způsobem (pomocí dočasných bufferů). Je-li stránka (uložená ve sdílené paměti) modifikována, nejprve se její předobraz zapíše do bufferu fyzického logu. Na disk se stránka zapíše až poté, co byl její předobraz z bufferu fyzického logu zapsán do fyzického logu na disk.

### 14.2 Logický log

**Logický log** udržuje informace o změnách od poslední archivace dat. Systémem OnLine je v reálném čase spravován jako nejméně tři samostatné oblasti na disku. Každá tato oblast vlastně tvoří jeden samostatný logický log. Logické logy jsou jedinečně očíslovány.

Záznamy logického logu používá systém OnLine k opravě a obnově všech transakcí, které proběhly od posledního stavu fyzické konzistence databáze (takový stav konzistence se označuje jako *checkpoint*), a slouží k obnově logické konzistence až po poslední záznam v logickém logu.

Dokud není soubor logického logu zazálohován nebo obsahuje otevřené transakce, je označen jako *used*, v opačném případě ho systém OnLine označí jako *free* a povolí jeho přepsání novými záznamy logického logu. Systém OnLine zapisuje transakce do logických logů ve vzestupném pořadí podle jejich jedinečných čísel. Je-li logický log zaplněn, systém OnLine se pokusí zapisovat do nejbližšího dalšího za předpokladu, že je tento označen jako *free*. Pokud je však logický log, do něž se má začít zapisovat, označen jako *used*, systém OnLine už dále nemůže normálně pokračovat.

Minimální počet souborů logického logu je 3. Může se totiž stát, že se jeden soubor zálohuje, do druhého se zapisují transakce a zaplní se dříve, než skončilo zálohování prvního souboru. Počet souborů logického logu udává parametr *LOGFILES* (minimální hodnota je 3), velikost každého souboru v KB parametr *LOGSIZE* (minimální hodnota je 200 KB).

Logický log obsahuje záznamy následujících typů:

- SQL definice dat všech databází
- záznamy o checkpointech

- záznamy o změnách konfigurace
- SQL dotazy, které mění stav databáze otevřené s logy
- pro konkrétní databázi záznam o změnách stavu logu

Při inicializaci systému OnLine se vytvoří soubory logických logů v oblasti *root dbospace*. Po přechodu systému OnLine do klidového stavu (*quiescent mode*) je možno několik souborů logického logu „dropnout“ z *root dbospace* a přidat je do jiného *dbospace*, čímž se může výrazně zlepšit výkon celého systému.

Faktory ovlivňující velikost a trvání jedné transakce:

- **Velikost záznamu logického logu** závisí na prostředí systému OnLine i na samotném provádění transakcí. Počet a délku generovaných záznamů logického logu ovlivňuje velikost tabulky, někdy i řádky. Záznam o checkpointu obsahuje tyto hodnoty pro každou transakci otevřenou při vytvoření checkpointu. Velikost tohoto záznamu má pak vliv na typ a úroveň aktivit aktuální databáze.
- **Doba otevření transakce** může být dost vysoká, což může způsobit chybu „dlouhá transakce“.
- **Počet a velikost aktivit CPU** ovlivňuje schopnost procesů systému OnLine dokončit transakci. Čas CPU, který každý proces potřebuje k provedení transakce, se zvyšuje s počtem zápisů do logického logu. Zvýšení aktivity v logickém logu může mít za následek větší boj o synchronizační prostředky logu (zámky). Proto může být výhodné přesunout soubory logického logu z *root dbospace* do jiného *dbospace*.
- **Frekvence rollbacku transakcí** ovlivňuje poměr, podle něž dochází k zaplňování logického logu.

Zápis *blobospace* do logického logu probíhá mírně odlišně: systém OnLine vyžaduje, aby se operace vytvářející *blobospace* a operace vkládající do něj *bloby* (*binary large objects*) zaznamenávaly do různých souborů logického logu. Přepnutí na další soubor logického logu se provádí příkazem **tbmode -l**.

### 14.3 Dlouhá transakce

**Dlouhá transakce** nastane, pokud se do souboru logického logu zapisuje za značku **LTXHWM** — *long-transaction high-water-mark*. V té chvíli démon **tbinit** začne vyhledávat otevřenou transakci v nejstarším používaném souboru logického logu. Pokud ji nalezne, nařídí rollback této transakce (nalezne-li více dlouhých transakcí, může nařídít více rollbacků najednou). Cílem je uvolnit nejstarší používaný soubor logického logu, než se v něm dosáhne kritického bodu **LTXEHWM** — *exclusive access long-transaction high-water-mark*, což bývá defaultně na 90 procentech souboru. V takovém případě se většině procesů

systemu OnLine zakáže přístup do aktuálního logického logu. Zapisovat do něj smějí pouze procesy provádějící rollback nebo commit. Nepovede-li se pomocí rollbacku transakci zrušit před zaplněním logického logu, dojde k shutdownu systému OnLine a je nutné zahájit obnovu dat. V jejím průběhu se však nesmí provést rollforward posledního souboru logického logu, jinak by se logický log opět zaplnil.

## 15 OnLine

### 15.1 Obnova dat

Rychla obnova (*fast recovery*) je automaticka vlastnost systemu OnLine. Provadi se pri zmene z modu *off-line* do modu *quiescent*. Jejim cilem je vratit OnLine system do stavu fyzicke a logicke konzistence s minimalnimi ztratami vykonane prace. Fast recovery zajistuje dve veci. Zaprvé je to pouziti fyzickeho logu k navratu k poslednimu fyzickemu konzistentnimu stavu, coz je posledni check-point. Zadruhé pouzije logicky log pro navrat k logicke konzistenci pomoci znovuzavedeni (*roll forward*) transakci potvrzenych od posledniho checkpointu a *rollback* transakci, které dokončeny nebyly.

Pri prechodu z *off-line* modu do modu *quiescent* OnLine kontroluje, zda je nutne fast recovery provadet. Pri inicializaci sdílené paměti *tbinit* kontroluje obsah fyzickeho logu. Pri kontrolovaném shut-downu systemu OnLine by měl být fyzický log prázdný. Přesun z OnLine modu do modu *quiescent* vyžaduje provedení checkpointu, který zajistí vyprázdnění fyzickeho logu. Pokud tedy fyzický log prázdný není, je jasné, že došlo k nekontrolovanému shutdownu a provede se fast recovery.

Fast recovery se skládá z následujících kroků:

- vrátí všechny diskové stránky do jejich stavu v průběhu posledního checkpointu pomocí dat z fyzickeho logu. Deje se tak pomocí zápisu předobrazu (*before-images*) uložených ve fyzickém logu. Každý *before-image* ve fyzickém logu obsahuje adresu stránky, která byla po checkpointu aktualizována.
- zjistí poslední záznam checkpointu v logickém logu. Je zaručeno, že takový záznam v logickém logu existuje. K jeho zjištění se využívá aktivní stránka *PAGE\_CKPT* z prostoru rezervovaných stránek *root dbspace*.
- provede všechny transakce potvrzené za posledním checkpointem.
- provede rollback všech nepotvrzených transakcí. Tím je zaručeno, že databáze bude v konzistentním stavu. Rollback může pokračovat i přes několik checkpointů, neboť i transakce mohla být bezet přes několik checkpointů. Je to umožněno tím, že žádný soubor logického logu, který obsahuje záznamy pro otevřené transakce, ještě nebyl uvolněn.

### 15.2 Procedura obnovy dat

Obnova dat se provádí při větších poruchách ze záložních kopií. Obsahuje následující kroky:

1. shromáždění všech archivních médií potřebných k obnově dat



2. nastavení parametru aktivní sdílené paměti na maximální hodnoty přiřazené od poslední archivace
3. kontrola, zda se konfigurace aktuálního zařízení shoduje s konfigurací při poslední archivaci
4. kontrola, zda jsou dostupná všechna potřebná zařízení
5. převedení OnLine do off-line módu
6. volba *Restore* z nabídky *Archiv* programu *DB-Monitor*, nebo spuštění *tb-tape -r*
7. přiřazení první archivní pásky úrovně 0 zařízení *TAPEDEV* příkazem *mount*
8. nyní proces *tb-tape* čte informace z pásky a kontroluje kompatibilitu se současnou konfigurací
9. pokud se *tb-tape* pta, potvrdit zalohování zbylých souborů logického logu z disku. Pasku přiřadit zařízení *LTAPEDEV* příkazem *mount*.
10. nyní proces *tb-tape* čte každou stránku z archivních pásek a zapisuje tyto stránky na adresy obsazené v jejich hlavičkách.
11. po uložení poslední archivní pásky se spustí proces
12. *tbinit*, který vymaže fyzický log, aby předeseř spustění *fast recovery*.
13. proces *tbinit* inicializuje sdílenou paměť.
14. proces *tb-tape* se pta, které logické logy se mají provést znovu. Je třeba přiřadit správnou pásku zařízení *LTAPEDEV*.
15. po ukončení tohoto *roll forwardu* zůstává systém OnLine v quiescent módu a *tbinit* vrátí řízení administrátorovi do programu *DB-Monitor*

K obnově dat jsou třeba všechny archivní pásky a pásky obsahující zálohy logických logů od poslední archivace. Potřebné pásky lze zjistit z programu *DB-Monitor* z nabídky *Status*, volby *Archive*.

Soubory logického logu, které zůstaly na disku a nebyly zazalohovány, mohou být stále zahrnuty do obnovy dat. OnLine se zeptá, zda má tyto soubory zalohovat na pásku, takže po obnově archivních pásek lze tyto záznamy znovu provést.

Behem obnovy dat nelze kvůli kompatibilitě konfigurace reinitializovat sdílenou paměť, přidávat chunks nebo zařízení pro pásky.

### 15.3 Checkpointy

Provedeni checkpointu se vyvola, pokud je nastaven priznak *checkpoint-requested*.

Checkpoint muze nastat v nekterem z nasledujich pripadu:

- je dosazena defaultni doba intervalu pro checkpoint (parametr *CKPTIN-TVL*) a od posledniho checkpointu doslo k nejakym zmenam.
- fyzicky log je zaplnen z 75 procent.
- OnLine zjist, ze nasledujici soubor logickeho logu, který ma zacit pouzivat, obsahuje posledni zaznam o checkpointu.
- administrator si provedeni checkpointu vyzada z DB-Monitoru nabidkou *Force-Ckpt*, nebo *tbmode -c*.
- *tbinit* nastavi priznak pri svem spusteni.

Behem checkpointu se fyzicky buffer zapise na disk, zapisou se vsechny modifikovane stranky, do bufferu logickeho logu je zapsan zaznam o checkpointu a buffer logickeho logu je zapsan na disk do aktualniho souboru logickeho logu.

Od okamziku nastaveni priznaku *checkpoint-request* nemohou uzivatelske procesy vstupovat do kritickych sekci. Procesum v kritickych sekcich je dovoleno pokracovat. Jakmile vsechny procesy kriticke sekce opusti, *tbinit* nastavi ukazatel na sdilenou pamet ze soucasneho bufferu fyzickeho logu na jiny buffer. Soucasny buffer zapise na disk. Pak *tbinit* aktualizuje strukturu popisujici sdilenou pamet tak, ze upravi casove razitko (*timestamp*), ktere zaznamenava posledni okamzik, kdy byl fyzicky buffer zapsan na disk. *Tbinit* nebo *tpgcl* (*page cleaner*) zapise vsechny modifikovane stranky sdilene pameti. Po zapisu stranek na disk je do bufferu logickeho logu zapsan zaznam *checkpoint-complete* a tento buffer je zapsan na disk. *Tbinit* zapise vsechny konfiguracni a archivacni informace do prislusne rezervovane stranky, at uz doslo ke zmene techto informaci nebo ne. Pokud dojde k pridani nebo k odebrani chunks nebo mirror chunks do/z oblasti dbspace, tyto zmeny jsou zaznamenany v oblasti sdilene pameti. Pokud od posledniho checkpointu doslo ke zmene techto informaci, *tbinit* zapise descriptoru techto stranek ze sdilene pameti do prislusne rezervovane stranky v oblasti root dbspace. Do prislusnych stranek jsou zapsany statisticke udaje. Pak *tbinit* uvolni nepotrebne soubory logickeho logu a zapise zaznam *checkpoint-complete* do logu zprav systemu OnLine.

### 15.4 Na co si dat pozor v prostredi OnLine

- Nikdy nepouzivat prikaz *kill* na OnLine server behem databazovych aktivit. Misto toho se doporučuje zkontrolovat, zda server nedrzi nejaky zamek a neni v kriticke sekci a po ukonceni procesu pouzit prikaz *tbmode -z*.

- Pokud možno nepoužívat transakce narocne na prostor logického logu.
- Nedefinovat zařízení archivní pasky (*TAPEDEV*) jako pojmenovanou rouru.
- Vytvářet záložní kopie databázi.
- Utility zapisující na pásku nepouštět na pozadí.
- Pokud se právě používají soubory logického logu, nepřepínat mezi zařízeními pasky logického logu a zařízením */dev/null*.
- Neumístovat chunky pro zrcadlení (mirror chunks) na stejné zařízení jako primární chunky.

## 16 Utility

### 16.1 Způsob uložení dat na disku

Informix buď přistupuje na disk pomocí služeb OS nebo přímým přístupem. V prvním případě je alokováno místo na disku pro jeden soubor, který představuje celý diskový prostor Informix. Práci s tímto souborem na disku má na starost OS, vnitřní organizace je pak vytvářena samotným Informixem. V druhém případě se o ukládání dat (a to jak fyzické rozmístění na disku, tak i logické uspořádání) stará pouze Informix.

Online systém pro ukládání dat používá pro popis fyzického diskového prostoru následující pojmy:

- **Chunk** — maximální prostor vyhrazený pro DBMS na jednom fyzickém disku.
- **Page** — jednotka udávající minimální počet bytů načtených při jednom přístupu na disk. Tato velikost je dána typem hardware počítače a není možné ji z uživatelského pohledu ovlivnit.
- **Blobpage** — alokační jednotka diskového prostoru, uchovávající datový typ BYTE nebo TEXT v Blobospace (viz níže). Její velikost je udána v násobcích Page a je stanovena administrátorem při vytváření Blobospace.
- **Extent** — jednotka udávající velikost fyzického diskového prostoru, který je alokován při vytváření nové tabulky. Pokud je tento prostor zaplněn, pak je alokován další prostor o velikosti jednotky Extent.

Tento fyzický prostor je pak rozdělen do následujících logických útvarů:

- **Dbospace** — při vytváření objektů databáze (tabulky atd.) určujeme Dbospace, ve kterém se bude příslušný objekt nacházet. To nám následně umožňuje ovlivnit způsob uložení těchto Dbospace na disku a tím tedy např. chránit důležitá data (např. jejich zálohováním nebo uložením na separátním disku atd.).

#### 16.1.1 Root Dbospace, Temporary Dbospace

- **Blobospace** — označení pro jeden či více Chunk, které uchovávají pouze datové typy TEXT nebo BYTE a to tím, pokud možno, nejefektivnějším způsobem. Bloby (TEXT nebo BYTE data) obsažené v různých tabulkách mohou být uloženy ve stejném Blobospace. Při ukládání jsou data zapsána přímo na disk, bez mezipřístupu do sdílené paměti a také z toho důvodu nejsou přístupy do Blobospace zaznamenávány do žurnálu, ale jsou zálohovány vcelku až při archivaci žurnálu.

- **Database** — logická jednotka, obsahující tabulky a indexy a případně také systémový katalog, nesoucí informace o jednotkách databáze (tzn. tabulky, indexy, uložené procedury a integritní omezení) Table - logicky je Table tabulka dle relačního databázového modelu, tedy obsahuje alespoň jeden sloupec a případně nějaké datové řádky, nesoucí hodnoty příslušných sloupců. Při vytváření tabulky je alokován extent (viz výše), jehož velikost (a to jak prvotního, tak i následných) je možno explicitně zadat.

### 16.1.2 Temporary Tables (explicit/implicit)

- **Tblspace** — jednotka popisující diskový prostor obsazený konkrétní tabulkou.

## 16.2 Utility pro diskovou administraci

### 16.2.1 oncheck

Umožňuje zobrazit strukturu rozložení dat na disku a detekovat v ní chyby. Při zjištění nekonzistence některého z indexů je možno pomocí této utility tyto chyby i odstranit.

Objekt	Check	Repair	Display
Blobspace blobs			-pB
Chunks & extents	-ce		-pe
Datové řádky bez DATA a TEXT	-cd		-pd
Datové řádky a Blobpages	-cD		-pD
Index (klíčové hodnoty)	-ci	-ci -y, -pk -y	-pk
Index (klíče a rowid)	-cI	-cI -y, -pK -y	-pK
Index (jen hodnoty klíčových listů)		-pl -y	-pl
Index (hodnoty klíčových listů a rowid)		-pL -y	-pL
Pages (dle tabulek)			-pp
Pages (dle chunků)			-pP
Stránky rezervované pro Root	-cr		-pr
Obsazené místo (dle tabulek)			-pt
Obsazené místo (dle tabulek, plus indexy)			-pT
Tabulky systémového katalogu	-cc		-pc

### 16.2.2 oninit

Slouží k inicializaci sdílené paměti, případně diskového prostoru, pokud jsou zadány příslušné volby. K vykonání těchto akcí je nutné být zalogován jako uživatel root nebo informix.

Volby:

- -p — tato volba způsobí, že při inicializaci nejsou vyhledány dočasné tabulky a celý proces je tedy o poznání rychlejší, ovšem s tou nevýhodou, že diskový prostor obsazený těmito dočasnými tabulkami není uvolněn.
- -s — inicializace sdílené paměti proběhne normálně, ale po jejím skončení je OnLine systém zanechán v neaktivním stavu.
- -i — proběhne inicializace sdílené paměti a diskového prostoru. Po skončení je systém v on-line módu. (Ve spojení s volbou -s je systém po inicializaci zanechán v neaktivním módu)

### 16.2.3 onload

Vytvoří tabulku ve specifikovaném Dbspace a naplní ji daty, které byly dříve uloženy na disk nebo zálohovací pásku pomocí utility onunload.

Volby:

Nejprve následují volby specifikující zdroj pro načtení dat do tabulky:

- -b blocksize — velikost bloku (kB) páskového zálohovacího zařízení
- -l — parametry páskového zálohovacího zařízení jsou načteny z příslušných systémových proměnných (LTAPEDEV, LTAPEBLK, LTAPE SIZE).
- -s tapesize — velikost dat (kB) uložených na pásce
- -t device/filename — mountpoint páskového zálohovacího zařízení/soubor nesoucí data tabulky

Následující volby jsou důležité při vytváření tabulky:

- -d dbspace — specifikuje databázový prostor, ve kterém bude tabulka vytvořena
- -i oldindex newindex — při vytváření tabulky je index oldindex přejmenován na newindex (při vyvolání onload z příkazové řádky je nutné specifikovat jméno tabulky)

### 16.2.4 onlog

Slouží k zobrazení obsahu žurnálu nejvíce pro ladící účely. Onlog je možné spustit v off-line módu, pak je přístupný pouze žurnál uložený na disku. V případě spuštění v on-line nebo i neaktivním módu jsou přístupné i žurnály ve sdílené paměti.

Volby:

Slouží k selekci dat ze žurnálu a k ovlivnění výsledného formátu, ve kterém je výstup zobrazen.

- -b — tato volba způsobí zobrazení blobpages z příslušného blobspace, který je uložen na záložní pásce logického žurnálu
- -d device — mountpoint záložní pásky logického žurnálu
- -n logid — způsobí zobrazení pouze položek vybraného žurnálu
- -l — výstupní formát bude long (obsáhlejší)
- -t tblspace\_num — omezí výstup na položky pouze z konkrétního tablespace
- -u username — omezí výstup pouze na akce vyvolané příslušným uživatelem
- -x transaction\_num — omezí výstup na konkrétní transakci

#### 16.2.5 onmode

Dle příslušných voleb je možné provést jednu z následujících akcí:

- měnit pracovní mód OnLine systému (tzn. on-line, neaktivní, off-line atd.)
- vyvolání kontrolního bodu
- změna uložení, případně rozšíření sdílené paměti
- změna žurnálu
- ukončení sezení/transakce atd.
- určení typu zálohy (replikace) dat
- přidání/odebrání virtuálního procesoru
- změna formátu dat na 5.0
- obnova souborů .infos

Pro provedení výše zmíněných akcí je nutné být zalogovaný jako root nebo informix

Volby:

- -y — automaticky odpoví „yes“ na všechny dotazy
- -k — přepnutí do off-line módu a odstranění sdílené paměti
- -m — přepnutí z neaktivního do on-line módu
- -s — omezí nové přístupy do databáze a po dokončení všech aktivních akcí se přepne do neaktivního módu

- -u — ukončí aktivní procesy a přepne do neaktivního módu, ovšem sdílená paměť zůstane nedotčena
- -c — způsobí zapsání kontrolního bodu do žurnálu
- -n — okamžitě ukončí vynucenou rezidentnost sdílené paměti
- -r — okamžitě si vynutí rezidentnost sdílené paměti
- -l — přepne na následující logický žurnál
- -z sid — ukončí konkrétní sezení
- -Z address — okamžitě ukončí provádění transakce na příslušné adrese ve sdílené paměti
- -d standard/primary *dbname*/secondary *dbname* — nastaví příslušný mód replikace dat tj. v prvním případě ukončí replikaci dat a ukončí spojení s ostatními servery, ve druhém a třetím případě nastaví příslušný server jako primární resp. sekundární
- -a seg\_size — přidá segment sdílené paměti o příslušné velikosti (kB)
- -p number — přidání/odebrání je zadáno pomocí volby +/-, parametry procesoru pak hodnotou number
- -b version — změna formátu databáze na 5.0 (jediná povolená hodnota)
- -R — při spouštění podpůrných utilit je využíván soubor \$INFORMIXDIR/etc/.infos.dbservername, který je touto volbou možné, při jeho smazání či porušení, obnovit

### 16.2.6 onparams

Umožňuje přidávat/mazat či měnit velikost a umístění logického žurnálu. Při vyvolání této utility je nutné, aby se databáze nacházela v neaktivním módu a uživatel musí být nalogován jako root nebo informix.

Volby:

- -y — odpověď „yes“ na všechny příslušné dotazy
- -a — přidá logický žurnál do OnLine systému
- -d dbspace — specifikuje databázový prostor, ve kterém bude nový žurnál uchován
- -s size — velikost (kB) nového žurnálu
- -d — označuje mazání log. žurnálu



- -l logid — určuje, který žurnál má být smazán
- -p — určuje změnu parametrů žurnálu
- -d dbspace — určuje umístění fyzického žurnálu v konkrétním databázovém prostoru
- -s size — určuje novou velikost žurnálu

### 16.2.7 onspaces

Umožňuje provádění následujících akcí:

- vytvoření blobspace, dbspace nebo dočasného dbspace
- smazání blobspace nebo dbspace
- přidání chunk
- smazání prázdného chunk
- spuštění/ukončení zrcadlení
- změna stavu chunk-u

Volby:

- -y — automatická odpověď „yes“
- -c — indikuje vytváření blobspace/dbspace
- -b blobspace — jméno vytvářeného blobspace
- -d dbspace — jméno vytvářeného dbspace
- -t — indikuje vytvoření dočasného dbspace
- -g page\_unit — velikost blobpage
- -m pathname offset — cesta a offset k chunku, který bude zrcadlit prvotní chunk nového blob/dbspace
- -o offset — posunutí počátku chunku na příslušném zařízení
- -p pathname — cesta k zařízení nesoucí prvotní chunk
- -s size — velikost prvotního chunku
- -d — indikuje mazání blob/dbspace (ve spojení se jménem příslušného objektu)

- -a — indikuje přidání nového chunku (ve spojení se jménem blob/dbspace, kterému bude nový chunk přiřazen)
- -d — indikuje mazání chunku (specifikace chunku viz volby výše)
- -m [pathname offset] — spustí zrcadlení konkrétního blob/dbspace (ve spojení se jménem příslušného objektu; volby viz výše)
- -f filename — specifikuje soubor, nesoucí informace o uložení chunku
- -r — ukončí zrcadlení konkrétního blob/dbspace
- -s — indikuje změnu stavu chunku (ve spojení se jménem blob/dbspace)
- -D — odstraní chunk ze systému
- -O — aktivuje chunk

#### **16.2.8 onstat**

Slouží ke sledování on-line operací. Utilita přečte obsah sdílené paměti a na základě jeho obsahu zobrazí statistiku požadované aktivity.

Akce či funkce	Volba
Zobrazení všech voleb	-
B-strom požadavků na uvolnění prostoru	-C
Vyrovnávací paměti (použité či nepoužité)	-B
Vyrovnávací paměti (použité)	-b
Vyrovnávací paměti (včetně adres čekajících vláken)	-X
Vyrovnávací pamět hašovacího řetězce	-h
Informace o konfiguračním souboru (\$INFORMIXDIR/etc/\$ONCONFIG)	-c
Dbospace chunks, všeobecné informace	-d
Dbospace chunks, pages reads/writes	-D
Interaktivní mód	-i
Závory	-s
Uzamčené zámky	-k
Informace o žurnálech (fyzické a logické žurnály, adresy obsazených stránek)	-l
LRU fronty	-R
Monitorovací informace	-g
OnLine žurnál zpráv	-m
Profil aktivity OnLine	-p
Opakuj shodný příkaz onstat periodicky	-r
Segment sdílené paměti (ulož do paměti)	-o
Přehled uživatelských voleb	-a
Tblspaces (aktivní)	-t
Informace o transakcích	-x
Uživatelská vlákna a transakce	-u
Statistika zápisu (získaná při uvolňování stránek)	-F
Vynulování všech statistických ukazatelů	-z

### 16.2.9 ontape

Zprostředkovává následující akce:

- archivace dat spravovaných databázovým serverem OnLine
- změna způsobu vedení žurnálu
- záloha logických žurnálů
- spuštění periodické zálohy logických žurnálů
- obnova zálohovaných dat z pásky
- využití replikace dat

Volby:

Příslušné volby jsou spojeny se jménem databáze, jíž se týkají.

- -s — inicializuje archiv
- -L archive\_level — určuje úroveň archivu
- -A — nastaví vedení žurnálu dle ANSI
- -B — nastaví vedení žurnálu s vyrovnávací pamětí
- -N — ukončí vedení žurnálu pro konkrétní databázi
- -U — nastaví vedení žurnálu bez vyrovnávací paměti
- -a — zálohuje všechny logické žurnály
- -c — nastaví kontinuální zálohu všech logických žurnálů
- -D — obnoví konkrétní databázi
- -r — obnoví celý systém ze zálohovacích pásky
- -l — obnoví všechny dbspace z příslušné repliky logického žurnálu
- -p — zahájí obnovu dat dle fyzického žurnálu (tato akce předchází volbě -l)

#### 16.2.10 onunload

Zapíše databázi nebo tabulku na do souboru na pásce nebo disku

Volby:

- -b blocksize, -s tapesize, -t device/filename — parametry páskového zařízení
- -l — vynucené načtení parametrů páskového zařízení z příslušných proměnných environmentu (LTAPEDEV, LTAPEBLK, LTAPESIZE)

## 17 Zámky

Zamykání je potřeba v případě, že bude 2 a více programů pracovat se stejnými daty. Např:

1. `select * from knihy`
2. `update knihy set cena = cena * 1.5`

Bez zamykání můžou nastat např. následující případy:

- můžu načíst některé řádky před `update` a některé až po `update`
- pokud je po `update` rollback, tak můžu načíst řádky, které do databáze nikdy nebyly commitnuty!!

### 17.1 Druhy zámků:

- **shared** — zamknutí pro čtení, provede se pouze pokud tam už není exclusive lock (jsou povoleny další shared locky)
- **exclusive** — zamknutí pro zápis, provede se pokud na daném objektu není žádný lock (nejsou povoleny žádné další locky)
- **promotable** — zamknutí s tím, že se zámek časem může změnit na exclusive (používá se, když mám kurzory „for update“), nesmí tam být žádný lock (jsou povoleny další shared locky)

### 17.2 Oblast působnosti zámků:

#### 17.2.1 Databáze

- při otevření databáze se aplikuje shared lock
- „DATABASE název EXCLUSIVE“ otevře databázi s exclusive lockem (používá se např. když budu dělat do databáze hodně změn a vím, že s ní nikdo jiný nepotřebuje pracovat)
- zámek platí do „CLOSE DATABASE“ či do změny databáze

#### 17.2.2 Tabulka

- zamyká se automaticky při „ALTER INDEX, ALTER TABLE, CREATE INDEX, DROP INDEX“, zámek se uvolní po skončení příkazu
- `LOCK TABLE table` explicitně zamkne tabulku (když v ní budu dělat hodně změn, tak je to rychlejší než zamykání jednotlivých řádků)

### 17.2.3 Řádka/stránka/klíč

- zamykají se implicitně např. při update, insert nebo delete
- jestli se zamykají řádky nebo diskové stránky se určuje v „CREATE TABLE“ nebo v „ALTER TABLE“
- update kurzor při čtení řádky nastaví promotable lock a při změně této řádky ho změní na exclusive
- při delete se dá lock na klíč smazané řádky, aby nešla vytvořit řádka se stejným klíčem dokud se ten delete nekomitne
- při čtení se aplikace shared locků řídí zvolenou Izolační úrovní.

## 17.3 Doba trvání zámků (závisí na tom, jestli se používají transakce)

### 17.3.1 tabulka

- transakce — do „COMMIT WORK“ (UNLOCK TABLE je zde neúčinné, LOCK TABLE musí být uvnitř transakce: po BEGIN WORK)
- bez transakcí — musí se explicitně ukončit „UNLOCK TABLE“

### 17.3.2 Řádek/stránka/klíč

- transakce — do „COMMIT WORK“
- bez transakcí — do té doby, než se daný řádek zapíše na disk

## 17.4 Izolační úrovně

Definují, jak se bude chovat implicitní zamykání, nakolik je můj program izolován od paralelně běžících programů. Nastavuje se příkazem „SET ISOLATION LEVEL TO ...“:

- **DirtyRead** — nic se nezamyká a ani žádné zámky netestuje, čte to, co je aktuálně na daném místě (mohu tedy číst i tzv. řádky-fantomy) — je to default při práci bez transakcí
- **CommittedRead** — nelze přečíst řádku, která nebyla potvrzena příkazem commit (před čtením řádky se testuje, jestli na ní není exclusive lock) — default při použití transakcí bez mode ANSI
- **CursorStability** — při práci s kurzory mám zajištěno, že mi nikdo nezmění řádku, se kterou právě pracuji (na čtenou řádku umístí podle typu kurzoru shared nebo promotable lock, když přecházím na novou řádku, tak zámeček uvolní a zamkne tu novou řádku)

- **RepeatableRead** — zajišťuje to, že při opětovném čtení řádky, kterou už jsem v dané transakci četl, dostanu stejná data (na každou čtenou řádku dá shared/promotable lock a uvolní ho až při „CLOSE CURSOR“ nebo při ukončení transakce) — standardní při mode ANSI

## 17.5 Lock mode

Nastavení chování programu, když přistupuje k zamknutým objektům, příkazem „SET LOCK MODE TO ...“:

- **wait** — čeká se na uvolnění zámku
- **not wait** — vrátí se chybové hlášení, že je objekt zamčený
- **wait 17** — čeká 17 s na uvolnění a po této době se vrátí s chybou

## 17.6 Demonstrační programy

Při praktiku byly zámky demonstrovány na programech p1.4gl resp. p1a.4gl, p2.4gl, p3.4gl, p4.4gl, které lze nalézt spolu s potřebnými definicemi formulářů v adresáři /home/riha/lock/pok a dále na programu Locking.4gl, který je rovněž s některými dalšími daty v /home/riha/lock/ref. Paralelně mají běžet p1 a p2 resp p3 a p4, Locking „pracuje samostatně“. p1, p2 a Locking vyžadují databázi (stores2t) s transakcemi, p3 a p4 databázi pokus bez transakcí, obě databáze lze vygenerovat programem sqldemo (tj. jsou to demonstrační databáze Informixu).

Demonstrační programy jsou taktéž připojeny k těmto skriptům.

## A Seznam autorů

Autor	O čem že to psal	Literatura
RNDr. Antonín Říha, CSc.	<b>Úvod</b>	
Slávek Rydval	<b>Formuláře</b>	[3]
Petr Václavek	<b>Generátor sestav ACE</b>	[3]
Lenka Grünwaldová	<b>Menu</b>	[neuedla]
David Šilar	<b>Informix-4GL</b>	[neuedl]
Tomáš Machalík	<b>4GL — menu, okna, formuláře</b>	[neuedl]
Jiří Kocanda	<b>Optimalizace dotazu</b>	[neuedl]
Stanislav Pavlíček	<b>Tvorba a užití uložených procedur</b>	[neuedl]
Ondřej Martínek	<b>INFORMIX SQL Triggers</b>	[4]
Marián Petráš	<b>Informix ESQ/L/C</b>	[neuedl]
Pavel Chromý	<b>Transakce v INFORMIXU</b>	[5, 3]
Pavel Chromý	<b>Pohledy v INFORMIXU</b>	[5, 3]
Vít Cvahoušek	<b>Bloby</b>	[neuedl]
Tomáš Machalík	<b>Fyzický a logický log</b>	[neuedl]
Lenka Grünwaldová	<b>OnLine</b>	[neuedla]
Stanislav Pavlíček	<b>Utility</b>	[neuedl]
RNDr. Antonín Říha, CSc. a Pavel Semerád	<b>Zámky</b>	[6, 7]

Poznámka: Referát o Blobech je převzán z minulého školního roku, neboť letošní přednášející referát nedodal.

## B Formuláře

Všechny pomocné soubory jsou v adresáři FORMULAR. Je tam pět souborů. Dva jsou SQL scripty a tři jsou definice formulářů.

## C Reporty

Příklady jsou v adresáři REPORT. Jedná se o dvě sestavy.

## D Zámky

Příklady k zámkům jsou v adresáři zámky.



## Reference

- [1] Informix-SQL User Guide
- [2] DB-Access User Manual
- [3] Martin Lipš, Yvona Permová, Martin Sláma: INFORMIX (vydala GRADA leta páně 1995)
- [4] Using Triggers – INFORMIX User Manual
- [5] Informix 4GL Refernce Manual version 4.00, Volume 1 a 2
- [6] Informix-4GL: Reference manual, kap.3
- [7] Informix Guide to SQL: Tutorial, kap.7

Napsáno za pomoci typografického systému  $\text{\LaTeX}$ .

(c) 1998-9 rk

Další odkazy:

- [rk@atrey.karlin.mff.cuni.cz](mailto:rk@atrey.karlin.mff.cuni.cz)
- <http://atrey.karlin.mff.cuni.cz/~rk>
- <http://atrey.karlin.mff.cuni.cz/~rk/informix.shtml>