KATEDRA INFORMATIKY
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
UNIVERZITA KOMENSKÉHO, BRATISLAVA

EFEKTÍVNE PROGRAMOVANIE V JAZYKU ABAP

Adrián Jágrik 2009

Efektívne programovanie v jazyku ABAP

DIPLOMOVÁ PRÁCA

Adrián Jágrik

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
KATEDRA INFORMATIKY

Študijný odbor 9.2.1 Informatika

Vedúci diplomovej práce
Ing. Michal Procházka

BRATISLAVA 2009

Čestne prehlasujem, že túto diplomovú prácu som vypracoval samostatne, len s použitím uvedenej literatúry a informačných zdrojov.

_____

Adrián Jágrik

Abstrakt

Autor:	Adrián Jágrik
Názov diplomovej práce:	Efektívne programovanie v jazyku ABAP
Škola:	Univerzita Komenského v Bratislave
Fakulta:	Fakulta matematiky, fyziky a informatiky
Katedra:	Katedra informatiky
Vedúci diplomovej práce:	Ing. Michal Procházka
Rozsah práce:	80 strán

Práca rozoberá problematiku efektívneho programovania v jazyku ABAP z hľadiska výkonnosti. V prvej časti ponúka prehľad nástrojov využiteľných pri tvorbe efektívnych programov a pri optimalizácii programov. V druhej časti pojednáva o rôznych špecifikách jazyka ABAP, porovnáva rôzne spôsoby a prináša konkrétne merania. Na základe ich výsledkov vyvodzuje závery.

Kľúčové slová

ABAP, Open SQL, výkonnosť

Abstract

Author:	Adrián Jágrik
Name of diploma thesis:	Effective programmming in ABAP language
University:	Comenius University, Bratislava
Faculty:	Faculty of Mathematics, Physics and Informatics
Department:	Department of Computer Science
Diploma thesis advisor:	Ing. Michal Procházka
Diploma thesis extent:	80 pages

Thesis analyses a problematic of performant programming in ABAP language. In a first part it provides a review of tools useful for creating performant programmes and for programmes optimising. In a second part it deals with various specifics of ABAP language, compares different approaches and brings concrete measurements. According to its results it draws conclusions.

Keywords

ABAP, Open SQL, performance

Predhovor

Vývoj ABAP programov v prostredí SAP je komplexný proces, ktorý do značnej miery ovplyvňuje výkonnosť aplikačných serverov a databázových serverov a následne môže ovplyvniť aj fungovanie biznis procesov zákazníka. Slabá výkonnosť môže viesť k zníženej použiteľnosti systému, k frustrácii používateľa alebo dokonca k nepoužiteľnosti procesu.

Veľké množstvo programov je písaných neefektívne. Existuje na to viacero dôvodov. Programy sú vyvíjané na vývojových systémoch, ktoré obsahujú veľmi malé množstvo dát v porovnaní s produktívnymi systémami, preto programátor pri spustení programu vníma dobu jeho trvania ako akceptovateľnú. Ďalej programátor nemusí byť zvyknutý využívať techniky a nástroje, ktoré sú dostupné až od novších verzií, alebo sú primárne určené administrátorom systému. Efektívne programovanie z hľadiska výkonnosti môže byť niekedy na úkor prehľadnosti kódu a napísanie efektívneho kódu môže byť časovo náročnejšie ako realizovanie najjednoduchšieho spôsobu. Neustále sa meniace požiadavky na funkčnosť tiež vedú k vzniku neefektívnych programov. Ďalšími dôvodmi sú tiež neznalosť efektívnych techník, neuvedomenie si dopadu neefektívneho kódu a nedostatočný čas venovaný analýze a testovaniu.

Tejto problematike som sa rozhodol vo svojej práci venovať, pretože som sa zúčastnil viacerých projektov ako programátor, kde bolo mojou úlohou optimalizovať už existujúce programy. Tiež som sa zúčastnil projektu ako BC konzultant, čo mi umožnilo spoznať rôzne nástroje určené primárne pre administrátorov, ale využiteľné aj pri práci programátora.

Obsah

1 Slovník termínov

ABAP (Advanced Business Application Programming) – programovací jazyk vyššej úrovne vytvorený spoločnosťou SAP, v ktorom je vytvorená väčšina SAP aplikácií.

ABAP Dictionary (alebo tiež **Data Dictionary**) – kompletne integrovaná súčasť ABAP Workbench slúžiaca na centrálnu správu všetkých dátových definícií.

ABAP Workbench – množina programov pre vývoj aplikácií pre SAP ERP.

Biznis proces – množina aktivít vedúca k výstupu, ktorý je hodnotný pre zákazníka.

ERP (Enterprise resource planning) – počítačový softvér používaný na správu a koordináciu všetkých zdrojov, informácií a funkcií potrebných pre fungovanie biznis procesov.

Funkčný modul – procedúra definovaná v skupine funkcií, ktorá môže byť volaná z ľubovoľného ABAP programu. Je to vlastne špeciálny typ ABAP programu.

Krátky dump (alebo **ABAP dump**) – prerušenie behu programu vyvolané chybou pri behu programu (runtime chybou).

Open SQL – časť jazyka ABAP obsahujúca podmnožinu štandardných SQL príkazov. Príkazy Open SQL sú databázovým rozhraním transformované na SQL dotazy príslušnej databázovej platformy.

OSS Note (alebo tiež **SAP Note**) – prostriedok servisnej podpory spoločnosti SAP slúžiaci na informovanie o riešeniach problémov a inštrukciách pre

korektúry. OSS notes sú súčasťou servisného portálu https://service.sap.com

SAP buffer – oblasť pamäte aplikačného servera slúžiaca pre dočasné uchovanie niektorých dát z databázy pre umožnenie rýchlejšieho prístupu k dátam a odľahčenie databázy.

SAP modul – aplikačná jednotka SAPu enkapsulujúca biznis procesy, dáta a úlohy súvisiace s rovnakou oblasťou (logistika, personalistika, účtovnictvo, skladové hospodárstvo,...).

SAP štandard – množina programov dodaná priamo spoločnosťou SAP.

Subrutina – procedúra, ktorá môže byť volaná z programu, resp. skupiny funkcií, v ktorej je definovaná.

Transakcia – konkrétna aplikácia v prostredí SAP. Má priradený kód transakcie, ktorý je jej spúšťačom, a väčšinou program, prípadne aj číslo obrazovky pri dialógových programoch.

User-exit – miesto v programe, ktoré umožňuje do štandardného programu dorobiť vlastnú funkcionalitu.

Užívateľské rozšírenie – rozšírenie štandardného programu o dodatočnú funkcionalitu. Zahŕňa napríklad user-exity, implicitné a explicitné body rozšírenia.

Work proces – komponent aplikačného servera vykonávajúci rôzne typy požiadaviek v závislosti od typu work procesu: dialógový, background, update, enqueue, spool. Jeden work proces môže súčasne vykonávať jednu požiadavku. Počet rôznych typov work procesov je určený parametrami systému.

Zákaznícky program – program, ktorý nie je súčasťou štandardnej dodávky produktu od firmy SAP.

2 Úvod

Spoločnosť SAP je treťou najväčšou spoločnosťou v oblasti informačných technológií; jej produkt SAP ERP je najrozšírenejším ERP produktom a je nevyhnutnou súčasťou fungovania biznis procesov veľkého množstva významných spoločností. Tieto biznis procesy sú naprogramované prevažne v jazyku ABAP.

Hlavným zdrojom informácií o problematike tvorby efektívnych programov v tomto jazyku je certifikované SAP školenie BC 490 (ABAP Performance Tuning). Toto školenie poskytuje základné rady (ktoré som, prirodzene, zhrnul v kapitole Základné pravidlá), avšak nezachádza do dostatočnej hĺbky – rady podáva ako dogmy bez vysvetlenia príčin a hlbších súvislostí, tiež sa nevenuje nástrojom a technikám prístupným až od novších verzií. Ďalším zdrojom poznatkov sú jedine špecializované internetové fóra pre vývojárov v jazyku ABAP, kde si vývojári vymieňajú svoje skúsenosti a riešenia konkrétnych problémov z praxe, informácie sú však roztrúsené, nesystematické, často povrchné, plné balastu a nezriedka i nepresné. Tento stav bol pre mňa podnetom a motiváciou k výberu tejto témy.

Cieľom mojej práce je jednak kompilácia už existujúcich informácii o efektívnom programovaní v jazyku ABAP ako aj vykonanie rôznych meraní na konkrétnych príkladoch a vyvodenie záverov.

SAP systémy sú založené na klasickej trojvrstvovej architektúre. Z hľadiska výkonnosti sú pre mňa pre účely tejto práce zaujímavé viaceré veci súvisiace s databázovou a aplikačnou vrstvou, napríklad čas potrebný na vyhodnotenie dotazu databázovým serverom, množstvo prenášaných dát, zaťaženie aplikačného servera či počet prístupov do databázy. Klientskou vrstvou sa v tejto práci nebudem zaoberať.

Samotná optimalizácia a efektívne programovanie sú dosť všeobecné pojmy, ktoré pokrývajú obrovskú oblasť; vo svojej práci sa preto budem zaoberať najmä špecifikami jazyka ABAP a nebudem sa venovať samotnej optimalizácii SQL dotazu, čo by bola príliš všeobecná téma.

V prvej časti stručne popíšem nástroje, ktoré som využil pri písaní tejto práce alebo pri práci na niektorom z projektov, na ktorých som participoval. Predstavím ich účel a možnosti využitia pri optimalizácii, resp. vývoji programu.

V druhej časti sa budem zaoberať samotným porovnávaním jednotlivých prístupov a techník. Keďže aj to je rozsiahla oblasť, ktorá mnohonásobne presahuje rozsah tejto práce, budem sa zaoberať najmä prípadmi, ktorým sa nevenuje jediné certifikované SAP školenie tematicky sa prekrývajúce s mojou prácou, BC 490 (ABAP Performance Tuning). Tiež sa nebudem venovať natívnemu SQL, ale iba proprietárnemu Open SQL, vynechám tiež logické databázy, ktoré sú zastaralé a budem sa zaoberať len transparentnými tabuľkami (teda nie clustrovými a poolovými).

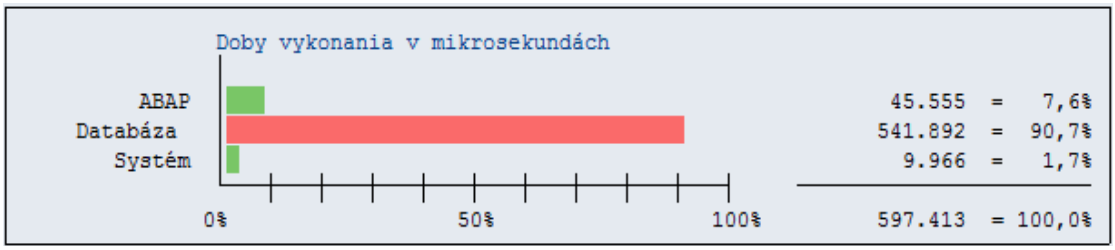
Merania som vykonával na rôznych systémoch, ale v tabuľkách uvádzam výsledky meraní na systéme bežiacom na HP Integrity rx6600 serveri s operačným systémom HP-UX 11i v2 pre Itanium2. Server má dva dvojjadrové procesory Itanium 2 a 12 GB operačnej pamäte. Aplikačný aj databázový server v tomto prípade bežia na tom istom serveri a dáta sú uložené na SAS diskoch. Použitý databázový server je v tomto prípade Oracle 10.2 Enterprise Edition 64-bit a použitá platforma aplikačného servera SAP Kernel 7.00 64-bit UNICODE, nad ktorou beží ERP aplikácia SAP ECC 6.0 SR2.

3 Nástroje využiteľné pri optimalizácii

Prostredie SAP ponúka viacero nástrojov využiteľných pri optimalizácii a meraní výkonnosti. Nástroje popísané v tejto kapitole som využil aj pri písaní tejto práce.

1.) ABAP príkaz GET RUN TIME FIELD rtime – pri prvom volaní od vytvorenia interného módu (t.j. spustenia programu) nastaví premennej rtime hodnotu 0. Pri každom ďalšom volaní sa nastaví hodnota premennej na dobu, ktorá uplynula od prvého volania. Ak pred ním nie je zavolaný príkaz SET RUNTIME CLOCK RESOLUTION LOW, meranie je vykonávané s presnosťou na mikrosekundy. To umožní zistiť dobu trvania ľubovoľného úseku užívateľského programu.

2.) Transakcia SE30 (Analýza chodu programu) – umožňuje analyzovať dobu behu programu. Ponúka sumárne výsledky, ktoré zobrazujú pomer doby trvania vykonávania príkazov na aplikačnej úrovni, na databázovej úrovni a na úrovni systémových volaní (načítanie programu do pamäte)

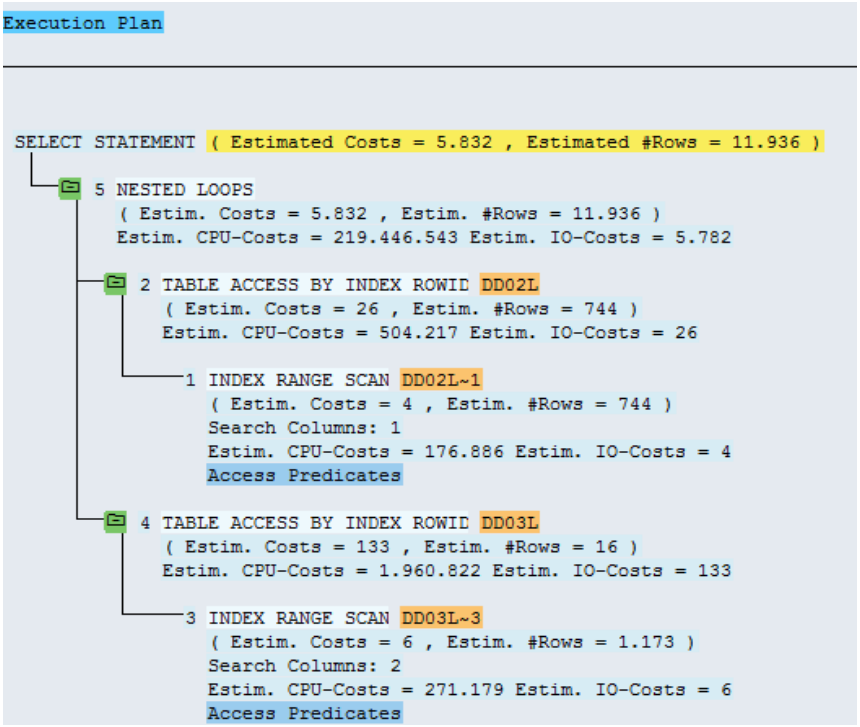


Ponúka tiež možnosť zobrazenia rebríčka, ktorý umožní identifikovať príkazy, ktoré trvajú najdlhšie, dobu ich trvania, ich počet volaní.

Poč.	Brutto	=	Netto	Brutto v %	Netto v %	Vyvolanie
1	597.413		0	100,0	0,0	Runtime analysis
1	590.588		287	98,9	0,0	Submit Report ZTEST_AJA
2	579.218		3.816	97,0	0,6	Program ZTEST_AJA
2	538.845	=	538.845	90,2	90,2	Fetch TADIR
1	269.858		138	45,2	0,0	Perform BUFFER_READ_USING_INDEX
1	269.421		26	45,1	0,0	Perform BYPASS_BUFFER_USING_INDEX
1	1.119	=	1.119	0,2	0,2	Fetch USR01
1	731	=	731	0,1	0,1	Select Single TRDIR
1	690	=	690	0,1	0,1	Open Cursor USR01
2	268	=	268	0,0	0,0	Open Cursor TADIR

Veľkou výhodou tejto transakcie je možnosť odskoku na ABAP kód, kde sa daný príkaz vykonáva.

3.) Transakcia ST05 (SQL Trace) – umožňuje zaznamenávať a vyhodnocovať SQL dotazy vykonané v databáze. Dotazy sú zobrazené ako natívne SQL dotazy databázového servera, nie ako Open SQL dotazy. To umožní vidieť, ako boli Open SQL príkazy interpretované databázovým rozhraním, a pri detailnom zobrazení je vidno aj to, ako boli realizované databázovým serverom (poradie joinov, využitie indexov). Tiež sú tu zobrazené ďalšie potrebné informácie, ako doba trvania a počet vrátených výsledkov. Ďalej je možné zobraziť identické dotazy aj spolu s počtom ich vykonaní. Pre každý natívny SQL príkaz je možnosť odskoku na príslušný Open SQL príkaz v ABAP kóde. ST05 tiež umožňuje sledovať prístup do buffrov, prípadne RFC volania a zamykanie a odomykanie objektov.



4.) Transakcia SLIN (Extended program check) a SCI (Code inspector) – nástroje na syntaktickú analýzu. Vykonávajú viaceré užitočné kontroly syntaxe programu ohľadom bezpečnosti, výkonu, zastaralých príkazov, nevyužívaných premenných a subrutín atď. Čo sa týka kontrol výkonu, vykonávajú sa v nich iba základné kontroly. Výhodou Code inspectora je možnosť implementácie vlastných testov a kontrol.

5.) Transakcia SM50 (Prehľad procesov) – umožňuje sledovať program počas jeho behu. Zobrazuje čas, ktorý program strávil vo work procese, aktuálnu akciu, prípadne tabuľku, ku ktorej program práve pristupuje. Je možné z neho odskočiť do ladenia programu (debuggera), čo umožňuje využiť v prípade potreby metódu Monte Carlo na identifikovanie problému.[1] Táto transakcia je využívaná skôr administrátormi na identifikovanie problému ako samotnému programátorovi.

6.) Transakcia DB05 (Analysis of table with respect to indexed fields) – slúži na analýzu selektivity polí jednotlivých tabuliek na základe záznamov z týchto tabuliek.

1 SCHWARZ, W. *Performance Problems in ABAP Programs: How to Find Them.*

7.) Transakcia ST10 (Performance analysis:Table call statistics) – ponúka štatistiku prístupov do databázových tabuliek a buffera. Tiež obsahuje informáciu o type použitia buffera a či je buffer správne synchronizovaný.

8.) Transakcia ST04 (Performance overview) – obsahuje viaceré štatistiky, ktoré sú však dôležitejšie pre administrátora ako pre programátora. Programátorovi poskytuje najmä dodatočnú funkciu „SQL Command Editor“, ktorá mu umožňuje vykonať Open SQL príkaz aj na systéme, ktorý nie je vývojový, a teda má možnosť analyzovať dobu jeho trvania na skutočných dátach na testovacích respektíve produktívnych systémoch. Táto funkcia je podporovaná iba pre databázovú platformu Oracle.

9.) Debugger – umožňuje krokovať program, dynamicky nastavovať breakpointy na príkazy jazyka ABAP, nastavovať watchpointy, zobraziť hodnoty premenných atď.

4 Základné pravidlá

Základne pravidlá pre prácu s dátami, vychádzajúce zo školenia BC490 a najbežnejších materiálov venujúcich sa problému výkonnosti:

1.) WHERE podmienka by mala byť čo najpodrobnejšie špecifikovaná. Teda množstvo vyhovujúcich záznamov pri selekcii by malo byť na čo najnižšej úrovni, aby sa zredukoval počet I/O operácií databázového servera a sieťové vyťaženie. Samozrejme, aj pri tomto pravidle platia výnimky. Ak WHERE podmienka predĺži výrazne dobu vyhodnocovania dotazu, je treba toto pravidlo posudzovať individuálne.

2.) Neprenášať zbytočne dáta medzi databázou a aplikačným serverom.

   a.) Využívať zoznam polí namiesto SELECT *. V prípade, že nás zaujímajú iba niektoré polia, ich špecifikovaním znížime množstvo prenášaných dát. Výnimkou sú poolové a clustrové tabuľky pri ktorých sú prenášané aj tak všetky polia.

   b.) Pre výber n-prvkovej množiny záznamov využívať UP TO n ROWS.

   c.) Pre UPDATE dát využívať UPDATE … SET príkaz ak je to možné namiesto updatovania z pracovnej oblasti alebo internej tabuľky, pretože pri UPDATE … SET sa prenáša zanedbateľné množstvo dát.

   d.) Používať agregačné funkcie Open SQL namiesto vykonávania agregačných funkcií na aplikačnej úrovni. (Toto pravidlo je možné využiť len pre transparentné tabuľky.)

   e.) Pri agregačných funkciách využívať HAVING podmienku.

3.) Zredukovať počet prístupov do databázy.

   a.) Pre operácie UPDATE, INSERT, DELETE, MODIFY využívať operácie nad internou tabuľkou namiesto individuálneho vykonávania operácie pre každý riadok.

   b.) Využivanie INNER a OUTER JOINov a poddotazov. To umožní zredukovať počet prístupov do databázy, prípadne aj množstvo

prenášaných záznamov. Pre často použité INNER JOINy je vhodné vytvoriť view.

4.) Optimalizovať dobu vyhľadávania vyhovujúcich záznamov.

a.) Využívať indexy.

b.) Využívať hinty v prípade, že databázový optimalizátor nenájde vhodný plán vykonávania.

5.) Odstrániť záťaž z databázy.

a.) Využívať buffer, ak je to možné.

b.) Nečítať tie isté dáta z databázy viackrát.

c.) V prípade UPDATE operácie skontrolovať, či je pred jeho vykonaním nutné naozaj vykonať SELECT, ak sa vykonáva.

d.) Vyhnúť sa triedeniu na databázovej úrovni, ak chceme triediť inak ako podľa indexových polí indexu, ktorý sa použije.

e.) Nepoužívať príkaz DISTINCT, ale namiesto neho využívať SORT a DELETE ADJACENT DUPLICATES.

6.) Pri subrutinách, funkčných moduloch a metódach používať pre interné tabuľky odovzdávanie referenciou a nie hodnotou.

5 Dynamické SQL príkazy

Základom dynamických Open SQL príkazov je použitie premenných alebo parametrov v SQL príkaze. Základným konceptom dynamického Open SQL je, že niektoré časti SQL príkazu nemusia byť vyjadrené staticky, ale obsahujú dynamické prvky, ktoré nie sú explicitne obsiahnuté v ABAP kóde. ABAP kód obsahuje premennú uzavretú v zátvorkách ako placeholder. Program musí obsahovať kód, ktorý vytvára ABAP zdrojový kód pre dynamické fragmenty a ukladá ho do premenných. Počas runtime, ABAP kód pre dynamické časti je parsovaný a zmiešaný so statickými časťami príkazu. Dynamické a statické časti sa javia databáze ako jeden segment. Tento proces je pre databázu kompletne transparentný. Dynamicky špecifikované môžu byť klauzuly FROM, INTO, WHERE, SELECT, GROUP BY, HAVING. Tiež môžu byť dynamicky vytvárané pracovné oblasti pre uchovanie dát. Dôvodom vzniku dynamického Open SQL bolo, že nie vždy bolo možné poznať všetky databázové tabuľky, tabuľkové polia alebo logické podmienky v dobe kompilácie programu. Dynamické Open SQL samozrejme má význam aj v prípade, že tieto veci sú známe aj počas kompilácie programu. Dôvody výhodnosti jeho použitia v takomto prípade môžu byť nasledovné:

- zjednodušenie a sprehľadnenie ABAP kódu – najmä ak potrebujem mať veľa podobných selektov líšiacich sa maličkosťami v rámci nejakého vetvenia. Okrem toho to zabezpečí jednoduchšiu údržbu programu, pretože zmena selektov sa nevykonáva na viacerých miestach, ale iba v mieste, kde sa samotné klauzuly vyskladávajú

- optimalizácia výkonnosti

Samozrejme, aj dynamické Open SQL má svoje nevýhody. Medzi ne patria napríklad:

- vytváranie dynamických príkazov je náročnejšie pre programátora a kód využívajúci dynamické vytváranie príkazov je menej prehľadný

- neexistuje možnosť kontroly syntaxe počas kompilácie, nesprávny kód môže spôsobiť ľahšie syntaktickú chybu počas runtimu. Z toho tiež vyplýva nemožnosť využitia Code inspectora na kontrolu optimálnosti takéhoto prístupu do databázy.

- pri runtime dochádza k dodatočnému parsovaniu ABAP programu, preto použitie dynamického Open SQL vnútri cyklu z výkonnostného hľadiska spôsobuje zdržanie. Viaceré merania, ktoré som vykonal s použitím rôznych dát a rôznych Open SQL príkazov, ukázali, že dodatočné náklady na parsovanie sa pohybujú od 0,01 ms po 0,04 ms pre jeden Open SQL príkaz s dynamicky špecifikovanými klauzulami v závislosti od počtu dynamicky špecifikovaných klauzúl a ich zložitosti. Toto zdržanie je zanedbateľné, pretože doba potrebná na parsovanie je v porovnaní s trvaním aj najrýchlejších dotazov zanedbateľná.

- debuggovanie dynamického Open SQL je náročnejšie

Možnosti dynamického špecifikovania jednotlivých klauzúl a možnosti dynamickej špecifikácie na získanie výkonnosti:

1.) SELECT – špecifikovanie podmienky pre projekciu. V prípade, že zoznam polí potrebných pre výber sa stáva známy až počas runtime, namiesto selektovania pomocou * je možné výkonnosť zlepšiť pomocou dynamického špecifikovania podmienky pre projekciu. To lineárne zredukuje množstvo prenášaných dát. Teda namiesto:

```
SELECT * FROM (gv_tabname) INTO CORRESPONDING FIELDS OF
TABLE lt_table.
```

je možné použiť:

```
  DATA:
  lt_field_list TYPE TABLE OF fieldname,
  lr_oref   TYPE REF TO cx_root,
  lv_text TYPE string.
```

```
… Naplnenie tabulky lt_field_list pozadovanymi
hodnotami...
  TRY.
      SELECT (lt_field_list) FROM (gv_tabname) INTO
CORRESPONDING FIELDS OF TABLE lt_table.
    CATCH cx_sy_dynamic_osql_semantics INTO lr_oref.
      lv_text = lr_oref->get_text( ).
      MESSAGE e398(00) WITH lv_text.
  ENDTRY.
```

2.) INTO – špecifikovanie cieľovej oblasti. V prípade, že zoznam polí potrebných pre výber sa stáva známy až počas runtime, namiesto určenia cieľovej tabuľky resp. cieľovej pracovnej oblasti, ktorá obsahuje všetky dovolené polia(teda obsahuje rovnaké polia ako databázová tabuľka) a využitia CORRESPONDING dodatku je možné cieľovú oblasť určiť dynamicky. To zredukuje náklady na dodatočné vyhľadávanie príslušných polí v cieľovej oblasti a tiež nároky na pamäť. Je možnosť buď iba dynamicky špecifikovať zoznam polí v cieľovej oblasti, alebo cieľovú oblasť aj dynamicky vytvoriť. K prvej možnosti nepovažujem za nutné uviesť príklad, funguje analogicky k dynamickému špecifikovaniu podmienky pre projekciu. Druhá možnosť sa dá v praxi využiť napríklad pri ALV reportoch, ktoré používatelia spúšťajú pomocou variantov, teda ich zaujímajú len niektoré polia. Rozšírim predchádzajúci príklad selekcie dát do internej tabuľky o dynamické špecifikovanie a vytvorenie cieľovej oblasti:

```
* i will use a class cl_abap_structdescr to get
components of Data dictionary object(structure or table)
  lr_ddic_struc_type ?=
cl_abap_structdescr=>describe_by_data( ls_ddic ).
  lt_comp_tab = lr_ddic_struc_type->get_components( ).
* i will find intersection of two sets: Data dictionary
```

```
components and requested fields
* Remark: I suppose that requested fields are subset of
data dictionary components
  LOOP AT lt_comp_tab INTO ls_comp.
    lv_index = sy-tabix.
    READ TABLE lt_field_list WITH TABLE KEY table_line =
ls_comp-name TRANSPORTING NO FIELDS.
    IF sy-subrc NE 0.
* If the component of data dictionary object is not
requested, i will delete it from components table
      DELETE lt_comp_tab INDEX lv_index.
    ELSE.
    ENDIF.
  ENDLOOP.

  IF lt_comp_tab IS INITIAL.
* No fields are requested, processing will not continue
    EXIT.
  ENDIF.

* I will create structure and table types dynamically,
using table of requested fields
  lr_struct_type =
cl_abap_structdescr=>create( lt_comp_tab ).
  lr_tab_type =
cl_abap_tabledescr=>create( lr_struct_type ).

* I will create internal table with dynamically specified
type
  TRY.
      CREATE DATA lr_dref_table TYPE HANDLE lr_tab_type.
    CATCH cx_sy_create_data_error INTO lr_oref.
      lv_text = lr_oref->get_text( ).
```

```
      MESSAGE e398(00) WITH lv_text.
  ENDTRY.
* ...and assign this table to field symbol
  ASSIGN lr_dref_table->* TO <lt_table>.

* I can perform select
  TRY.
      SELECT (lt_field_list) FROM (gv_tabname) INTO TABLE
<lt_table>.
    CATCH cx_sy_dynamic_osql_semantics INTO lr_oref.
      lv_text = lr_oref->get_text( ).
      MESSAGE e398(00) WITH lv_text.
  ENDTRY.
```

Čítanie a spracúvanie riadkov tejto internej tabuľky riadok po riadku by potom vyzeralo nasledovne:

```
  CREATE DATA lr_dref_structure TYPE HANDLE
lr_struct_type.
  ASSIGN lr_dref_structure->* TO <ls_structure>.
  LOOP AT <lt_table> INTO <ls_structure>.
    LOOP AT lt_comp_tab INTO ls_comp.
      CONCATENATE '<ls_structure>-' ls_comp-name INTO
lv_field_name.
      ASSIGN (lv_field_name) TO <lv_field>.
      CASE ls_comp-name.
        WHEN 'FIELD1'.
          .
          .
          .
        WHEN 'FIELD2'.
          .
          .
```

```
        .
      ENDCASE.
    ENDLOOP.
  ENDLOOP.
```

Viaceré merania ale ukázali, že prínos dynamického vytvorenia cieľovej oblasti z výkonnostného hľadiska nie je výrazný. Dôvodom je aj to, že výkonnostné zlepšenie sa týka iba aplikačnej úrovne a nie databázovej, a tiež, že na aplikačnej úrovni sa vykonávajú dodatočné operácie. Teda hlavnou motiváciou pri využití dynamického vytvárania cieľovej oblasti nebude zlepšenie výkonnosti.

3.) FROM – špecifikovanie mena tabuľky pre výber dát. Môže sprehľadniť kód, ale možnosti zlepšenia výkonnosti neprináša.

4.) WHERE – dynamické špecifikovanie podmienky pre projekciu dát. Dynamické špecifikovanie WHERE podmienky má vo všeobecnosti väčší význam pri zjednodušení a sprehľadnení kódu ako pri zlepšovaní výkonnosti. Avšak teoreticky je možné ho využiť aj na zlepšenie výkonnosti. Napríklad môže umožniť využitie IN operátora namiesto NOT IN operátora v prípadoch, v ktorých to nie je možné bez využitia dynamického určenia WHERE podmienky. Konkrétne ide napríklad o prípad, že máme obmedzenie na nerovnosť na kľúčové pole (resp. iné pole, ktoré je súčasťou indexu, ktorý bude použitý pri vyhľadávaní), ktoré má kontrolnú tabuľku. To by mohlo umožniť lepšie využitie indexu pri vyhľadávaní. Konkrétna myšlienka je nahradenie

```
    LOOP AT gt_tab1 ASSIGNING <gs_tab1>.
      SELECT * FROM db_table INTO TABLE gt_tab2 WHERE
key1 EQ <gs_tab1>-field1 AND key2 NOT IN (field1, field2,
field3,...).
    ENDLOOP.
```

nasledujúcim kódom:

```
  lv_in = 'key2 IN ( '.
* IN restriction will be dynamicaly concatenated from
values from a check table
  SELECT field INTO lv_field FROM db_check_table WHERE
field NOT IN (field1, field2, field3,...).
    REPLACE ',#' WITH ', ' INTO lv_where.
    CONCATENATE lv_in '''' lv_field ''',#' INTO lv_in.
  ENDSELECT.
  IF sy-subrc EQ 0.
* Select will be performed only if some value exist in a
check table
    REPLACE ',#' WITH ')' INTO lv_where.

    LOOP AT gt_tab1 ASSIGNING <gs_tab1>.
      SELECT * FROM db_table INTO TABLE gt_tab2 WHERE
key1 EQ <gs_tab1>-field1 AND (lv_in).
    ENDLOOP.
  ENDIF.
```

Napriek rôznym obmenám spôsobu realizovania tejto myšlienky v praxi a meraniach na rôznych systémoch, tabuľkách a indexoch sa mi nepodarilo týmto spôsobom nikdy dosiahnuť zlepšenie výkonnosti, vždy som dosiahol opačný efekt. Pravdepodobne sú databázové systémy v súčasnosti dostatočne vyspelé, aby dokázali dotaz obsahujúci negovaný IN operátor efektívne zrealizovať a efektívne využiť index.

5.) GROUP BY a HAVING – špecifikovanie podmienky pre agregáciu. V prípade komplikovanej agregačnej podmienky, ktorú nemožno určiť staticky, umožní nahradiť agregáciu na aplikačnej úrovni agregáciou na databázovej úrovní, čo môže viesť k zlepšeniu výkonnosti.

6.) UPDATE .. SET … WHERE – aj SET podmienka môže byť špecifikovaná dynamicky, rovnako aj meno tabuľky a WHERE podmienka pri UPDATE operácii. V niektorých prípadoch dynamické určenie WHERE a SET podmienky umožní použiť UPDATE pomocou SET dodatku aj tam, kde to ináč nie je možné. UPDATE pomocou dodatku SET je vo všeobecnosti rýchlejší ako iné spôsoby UPDATE operácie.

Ako vidno, aj keď je dynamické Open SQL určené najmä na to, aby umožnilo generický prístup k dátam, je možné ho využiť aj na zlepšenie výkonnosti.

6 Použitie kurzora pri práci s internými tabuľkami

V praxi sa často využíva vnorený loop, napríklad pre prípad, že máme dve interné tabuľky: tabuľku s riadkami hlavičky a tabuľku s riadkami položiek. Teda sa loopuje nad riadkami hlavičky a pre každú hlavičku sa spracúvajú príslušné položkové riadky. Vo veľkom množstve prípadov ide o tabuľky s veľkým a neustále rastúcim počtom riadkov, čo po čase spôsobuje veľké výkonnostné problémy, pretože vnorené cykly nad tabuľkami s veľkým počtom riadkov sú obrovskou záťažou pre aplikačný server. Najčastejší spôsob, ako sa spracúvajú 2 interné tabuľky, pričom majú spoločnú časť kľúča a k jednému záznamu z hlavičkovej tabuľky prislúcha niekoľko záznamov z položkovej tabuľky, vyzerá približne nasledovne:

```
LOOP AT gt_header INTO gs_header.
* processing of header
 LOOP AT gt_items INTO gs_items
     WHERE key1 EQ gs_header-key1
       AND key2 EQ gs_header-key2
       AND key3 EQ gs_header-key3.
* processing of item lines
    ENDLOOP.
  ENDLOOP.
```

Tento spôsob je najzdĺhavejší najmä v prípade, že tabuľka položiek je zadeklarovaná ako štandardná tabuľka, nie ako sortovaná tabuľka. Tento spôsob sa dá výrazne urýchliť použitím indexu alebo kurzora. Podstatou spôsobu využívajúceho index je, že dáta v tabuľke položiek utriedime podľa kľúčových polí a pri prechode na nový riadok hlavičky zistíme index prvého riadku s rovnakými hodnotami týchto polí v položkovej tabuľke. Potom nad touto tabuľkou loopujeme, až kým sa nelíši niektoré z kľúčových polí oproti hodnotám z hlavičky. Vtedy vieme, že sme spracovali všetky záznamy prislúchajúce hlavičkovému záznamu. V praxi to vyzerá približne nasledovne:

```
  SORT gt_items BY key1 key2 key3.
  LOOP AT gt_header INTO gs_header.
* processing of header
    READ TABLE gt_items TRANSPORTING NO FIELDS
        WITH KEY key1 = gs_header-key1
                 key2 = gs_header-key2
                 key3 = gs_header-key3
        BINARY SEARCH.
    IF sy-subrc = 0.
      lv_tabix = sy-tabix.
      LOOP AT gt_items INTO gs_items FROM lv_tabix.
        IF    gs_items-key1 NE gs_header-key1
           OR gs_items-key2 NE gs_header-key2
           OR gs_items-key3 NE gs_header-key3.
          EXIT.
        ENDIF.
* processing of items
      ENDLOOP.
    ENDIF.
  ENDLOOP.
```

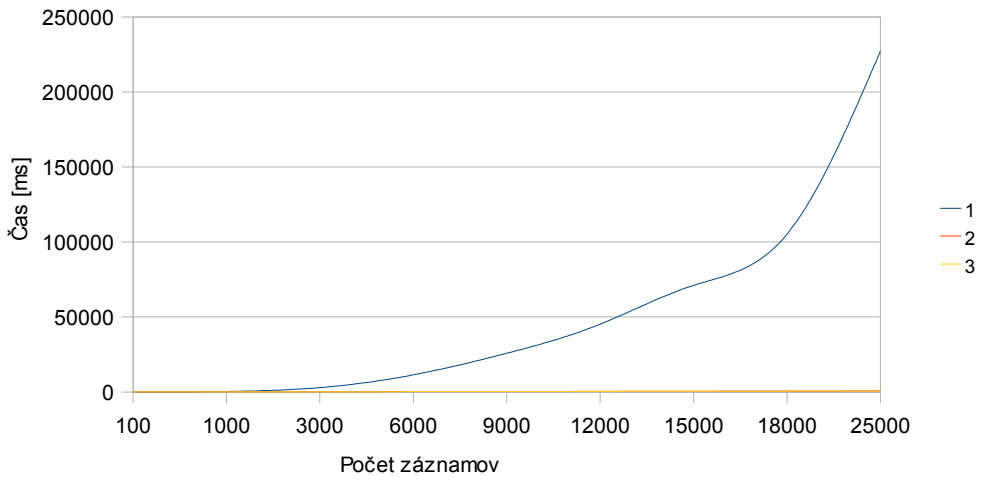
Pri využití kurzora sa vyžaduje utriedenie nielen tabuľky s položkami, ale aj hlavičkovej tabuľky. Podstatou tejto metódy je, že prechádzame obe tabuľky iba raz, keďže si uchovávame kurzor na riadok v položkovej tabuľke. (V príklade predpokladám, že ku každému hlavičkovému riadku prislúcha aspoň jeden riadok v tabuľke položiek.) V praxi môže byť tento spôsob realizovaný nasledovným kódom:

```
 SORT gt_header BY key1 key2 key3.
  SORT gt_items BY key1 key2 key3.
  lv_tabix = 1.
```

```
  LOOP AT gt_header INTO gs_header.
* processing of header
    LOOP AT gt_items INTO gs_items FROM lv_tabix.
      IF    gs_items-key1 NE gs_header-key1
         OR gs_items-key2 NE gs_header-key2
         OR gs_items-key3 NE gs_header-key3.
        lv_tabix = sy-tabix.
        EXIT.
      ELSE.
* processing of items
      ENDIF.
    ENDLOOP.
  ENDLOOP.
```

Počet záznamov\spôsob	1	2	3
100	7	3	2
1000	337	29	27
3000	2900	91	80
6000	11423	185	162
9000	25800	280	244
12000	45192	380	331
15000	71106	483	417
18000	105276	575	503
25000	227436	815	715

Tabuľka č.1 - Doby trvania behu jednotlivých spôsobov vzhľadom na počet záznamov v milisekundách.



Ako vidno z výsledkov merania, aj využitie indexu, aj využitie kurzoru priniesli rapídne skrátenie doby behu programu, keďže doba behu pri tomto spôsobe rastie lineárne s rastúcim počtom záznamov. Doby trvania sú pri oboch spôsoboch približne rovnaké a pri rozhodovaní sa medzi týmito dvoma spôsobmi treba zvážiť dve veci:

1.) či každý záznam v hlavičkovej tabuľke má príslušný záznam v tabuľke položiek – ak nie, je potrebné použiť spôsob využívajúci index, aby sa spracovali všetky položkové riadky

2.) aký je priemerný počet položkových riadkov prislúchajúcich k riadku hlavičky. Čím je toto číslo väčšie, tým je rozdiel v dobe behu medzi týmito dvoma spôsobmi menší, pretože operácia READ sa pri spôsobe s indexom volá v porovnaní s vnútorným LOOPom menejkrát. Avšak aj v prípade, že pripadá na jednu hlavičkovú tabuľku málo riadkov položiek, rozdiely v dobách behu sú pre tieto dva spôsoby zanedbateľné.

7 Používanie symbolov polí

Symboly polí sú symbolické mená pre ostatné polia. Fyzicky nerezervujú pamäťové miesto pre pole, ale ukazujú na jeho obsah. Môžu ukazovať na ľubovoľný dátový typ. Vždy, keď sa v programe pristupuje k symbolu poľa, pristupuje sa vlastne k poľu, ktoré je k nemu priradené - po úspešnom priradení symbolu poľa k poľu teda nie je žiaden rozdiel, či sa v programe pristupuje k symbolu poľa alebo k poľu samotnému. Aby však bolo možné pracovať so symbolom poľa, musí byť vždy priradený k nejakému poľu.[2]

Symboly polí môžu byť vytvorené so špecifikovaním typu alebo bez. Ak nie je špecifikovaný typ, symbol poľa zdedí všetky technické atribúty poľa k nemu priradenému. Ak je špecifikovaný typ, systém kontroluje kompatibilitu symbolu poľa a poľa, ktoré sa k nemu priraďuje počas príkazu ASSIGN.

Hlavnou výhodou symbolov polí je flexibilita. Najväčšou nevýhodou je, že použitie symbolov polí môže viesť k chybám, ktoré sa neodhalia počas kontroly syntaxe, ale až počas behu programu, prípadne môže viesť k zlým dátovým priradeniam. Použitie symbolov polí môže mať prínos aj pri zlepšení výkonnosti.

Využitie symbolov polí ako pracovnej oblasti pri práci s internými tabuľkami

Pri čítaní, resp. loopovaní nad internými tabuľkami sa bežne používa príkaz INTO a modifikácia jednotlivých riadkov vyzerá približne nasledovne:

1.)

```
  LOOP AT gt_table INTO gs_work_area.
   gv_tabix = sy-tabix.
* determine a value of a variable gv_new_value
   gs_work_area-field1 = gv_new_value.
```

2 http://help.sap.com/saphelp_nw04/helpdata/en/fc/eb3860358411d1829f0000e829fbfe/content.htm

```
   MODIFY gt_table FROM gs_work_area INDEX gv_tabix.
  ENDLOOP.
```

V tomto prípade sa pri spracovaní každého riadku obsah celého riadku skopíruje do pracovnej oblasti, potom sa zmení hodnota polí v pracovnej oblasti a na záver sa obsah riadku v tabuľke zmodifikuje na základe obsahu pracovnej oblasti; čiže je potrebná dodatočná pamäť na uloženie obsahu riadku do pracovnej oblasti a samotné kopírovanie obsahu sa vykonáva dvakrát, raz pri napĺňaní pracovnej oblasti a raz pri modifikovaní obsahu tabuľky. Mierne zlepšenie z výkonnostného hľadiska dokážeme dosiahnuť, ak pri modifikovaní tabuľky nebudeme prenášať obsah celého riadku, ale iba zmenené polia pomocou príkazu TRANSPORTING. Upravený kód by potom vyzeral nasledovne:

2.)

```
  LOOP AT gt_table INTO gs_work_area.
   gv_tabix = sy-tabix.
* determine a value of a variable gv_new_value
   gs_work_area-field1 = gv_new_value.
   MODIFY gt_table FROM gs_work_area INDEX gv_tabix
TRANSPORTING field1.
  ENDLOOP.
```

Ak sa chceme vyhnúť kopírovaniu obsahov polí úplne, môžeme na to využiť symboly polí a príkaz ASSIGNING. V prípade, že špecifikujeme typ symbolu poľa, kód by mohol vyzerať približne nasledovne:

3.)

```
  LOOP AT gt_table ASSIGNING <gs_work_area>.
* determine a value of a variable gv_new_value
    <gs_work_area>-field1 = gv_new_value.
  ENDLOOP.
```

V tomto prípade sa pri spracovaní každého riadka priradí symbolu poľa

ukazovateľ na riadok tabuľky, a teda žiadne kopírovanie jeho obsahu sa nevykonáva ani pri jeho načítaní do pracovnej oblasti, ani pri modifikácii internej tabuľky. Samotný kód vyzerá tiež jednoduchšie a prehľadnejšie.

Ďalšou možnosťou je použiť symbol poľa s nešpecifikovaným typom:

4.)

```
  LOOP AT gt_table ASSIGNING <gs_work_area>.
* determine a value of a variable gv_new_value
    ASSIGN COMPONENT 'FIELD1' OF STRUCTURE <gs_work_area>
TO <gv_field>.
    <gv_field> = gv_new_value.
  ENDLOOP.
```

V tomto prípade sa pri čítaní riadku internej tabuľky síce nekopírujú žiadne dáta, ale nie je možné priamo modifikovať obsah riadku modifikovaním pracovnej oblasti reprezentovanej symbolom poľa, pretože ako netypizovaný symbol poľa nemá žiadnu štruktúru, teda nie je možné modifikovať jeho komponenty. Samotné priradenie novej hodnoty je možné až po priradení komponentu štruktúry do ďalšieho symbolu poľa.

Porovnanie jednotlivých prístupov na základe merania prinieslo tieto výsledky:

Počet záznamov\spôsob	1	2	3	4
100	0,09	0,09	0,05	0,28
1000	0,82	0,84	0,45	2,77
50000	48,4	48,92	22,67	138,89
100000	98,02	101,88	44,42	278,99
300000	291,16	307,97	136,53	837,32
600000	586,8	619,93	276,88	1669,88
1200000	1179,85	1123,76	547,06	3342,31
2400000	2411,82	2462,51	1091,06	6696,22
4800000	4735,82	4942,57	2152,36	13390,74

Tabuľka č.2 - Doby trvania behu jednotlivých spôsobov vzhľadom na počet záznamov v milisekundách.

Ako ukazujú namerané výsledky, rozdiely medzi prvými troma spôsobmi nie sú nejako výrazne priepastné a pri množstve záznamov 2,4 milióna bol časový rozdiel len 1,4 sekundy. Najpomalší aj najrýchlejší spôsob boli s využitím symbolov polí. Najpomalší spôsob bol pri využití netypizovaného symbola poľa. Dôvodom bolo dedenie atribútov počas behu programu rovnako aj dodatočné priradenia komponentu štruktúry do ďalšieho symbolu poľa. Tento spôsob sa oplatí využívať, iba ak program vyžaduje naozaj takú flexibilitu, akú ponúka jedine netypizovaný symbol poľa. Pri využití typizovaného symbolu poľa sa dosiahli najlepšie výsledky, ako sa aj očakávalo, pretože sa nekopírovali obsahy riadkov. Spôsob č.2 využívajúci

dodatok TRANSPORTING k spôsobu č.1 nepriniesol podľa očakávaní výkonnostné zlepšenie, ale dokonca mierne zhoršenie. Dôvodom je pravdepodobne to, že vyhľadanie komponentu v štruktúre riadku je časovo mierne náročnejšia operácia ako skopírovanie obsahu celého riadku.

Záverom tohto merania je, že využitie typizovaných symbolov polí prináša najjednoduchší a najpochopiteľnejší kód a tiež najlepšie výsledky z hľadiska výkonnosti.

8 SELECT INTO TABLE vs. SELECT … ENDSELECT

Pri selekcii dát je možné ako cieľovú oblasť zvoliť buď internú tabuľku alebo štruktúru, ktorá slúži ako pracovná oblasť a po samotnom spracovaní riadku sa pridáva na koniec tabuľky. Ak použijeme selekciu do internej tabuľky, na dodatočné spracovanie riadkov bude potrebný dodatočný loop nad riadkami tabuľky. Možné dva spôsoby teda vyzerajú nasledovne:

1.)

```
  SELECT * FROM (gv_tabname) INTO gs_line.
* line processing
    APPEND gs_line TO gt_table.
  ENDSELECT.
```

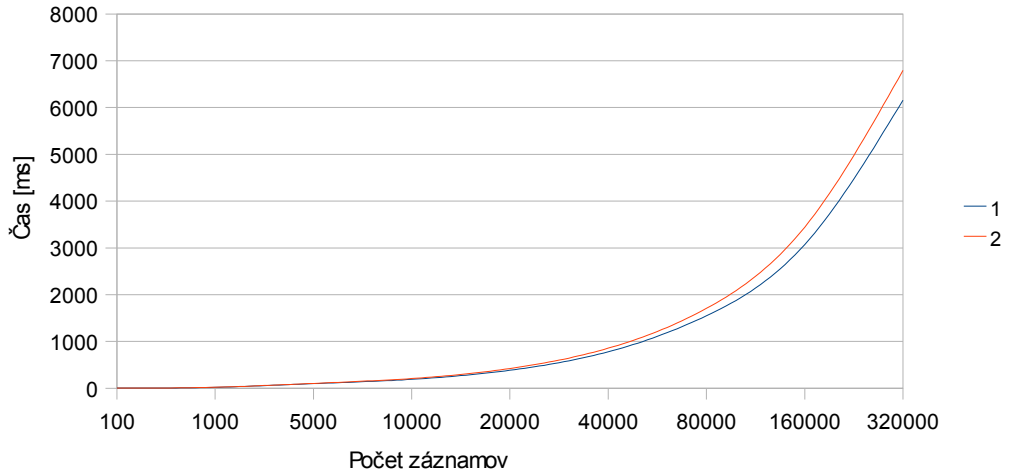
2.)

```
  SELECT * FROM (gv_tabname) INTO TABLE gt_table.
  LOOP AT gt_table ASSIGNING <gs_line>.
* line processing
  ENDLOOP.
```

Merania priniesli takéto výsledky:

Počet záznamov\spôsob	1	2
100	2,66	2,32
1000	19,19	20,47
5000	98,14	103,26
10000	189,19	206,93
20000	384,88	421,42
40000	777,83	855,26
80000	1547,74	1705,5
160000	3071,24	3444,41
320000	6158,34	6792,62

Tabuľka č.3 - Doby trvania behu jednotlivých spôsobov vzhľadom na počet záznamov v milisekundách.



Ako vidno z nameraných výsledkov, rýchlejší je spôsob využívajúci internú tabuľku ako cieľovú oblasť. Rozdiel vo výkonnosti je však malý až zanedbateľný a s rastúcim počtom záznamov sa neprehlbuje nijako rapídne. Oveľa väčšou motiváciou v uprednostňovaní selekcie do internej tabuľky pred selekciou dát do štruktúry je, že samotné spracovanie riadkov sa pri použití spôsobu č.2 vykonáva počas toho, čo už je databázový kurzor uzavretý. Pri spracovaní pomocou spôsobu č.1 je databázový kurzor počas spracovania riadkov stále otvorený. To môže vyvolať výnimku CX_SY_OPEN_SQL_DB a následný krátky dump. To programátora obmedzuje pri spracovaní riadkov a neumožňuje mu použiť nasledovné príkazy:

- MESSAGE (okrem MESSAGE S...)
- COMMIT WORK
- ROLLBACK WORK
- BREAK-POINT
- WAIT
- CALL FUNCTION ... DESTINATION (synchrónne RFC)
- CALL FUNCTION ... STARTING NEW TASK

- RECEIVE RESULTS
- CALL DIALOG
- CALL SELECTION-SCREEN
- CALL TRANSACTION
- CALL SCREEN alebo iný príkaz, ktorý vedie k zobrazeniu inej obrazovky

9 Rôzne typy interných tabuliek a ich využitie

V programovacom jazyku ABAP je použitie interných tabuliek veľmi časté. Interné tabuľky umožňujú uložiť dáta fixnej štruktúry v pamäti aplikačného servera. V ABAPe interné tabuľky tiež spĺňajú funkciu polí. Hoci sú to dynamické objekty, programátor sa nemusí starať o dynamickú správu pamäte vo svojom programe. Interné tabuľky sa využívajú pri spracovaní dát na aplikačnej úrovni. Mali by byť použité vždy pri spracúvaní nejakej množiny dát s fixnou štruktúrou. Umožňujú počas behu programu uchovávať a spracúvať dáta získané z databázy. Tiež sú vhodným a najčastejším spôsobom, ako obsiahnuť komplikované dátové štruktúry v ABAP programe. Podobne ako všetky prvky v koncepte typov jazyka ABAP, interné tabuľky môžu existovať aj ako dátové typy aj ako dátové objekty. Dátový typ je abstraktný popis internej tabuľky, či už v programe alebo centrálne v ABAP dictionary, ktorý sa používa na vytvorenie dátového objektu. Dátový typ je tiež atribút existujúceho dátového objektu. Dátový typ internej tabuľky je plne špecifikovaný typom riadku, kľúčom a typom tabuľky.

Typ riadku

Typ riadku internej tabuľky môže byť ľubovoľný dátový typ. Väčšinou to býva štruktúra, ale môže to byť aj elementárny dátový typ alebo iná interná tabuľka.

Kľúč

Kľúč identifikuje riadky tabuľky. Pre interné tabuľky existujú dva druhy kľúčov: štandardný kľúč a používateľsky definovaný kľúč. Programátor môže určiť, či má kľúč byť UNIQUE (unikátny) alebo NON-UNIQUE (neunikátny). Interné tabuľky s unikátnymi kľúčmi nemôžu obsahovať duplikátne záznamy. Unikátnosť záleží na prístupovej metóde k tabuľke.

Pokiaľ má tabuľka štruktúrovaný typ, jej defaultný (predvolený) kľúč pozostáva zo všetkých nenumerických stĺpcov, ktoré nie sú referencie alebo

interné tabuľky. Ak je typ riadku tabuľky elementárny, kľúčom je celý riadok. Predvolený kľúč internej tabuľky, ktorej typom riadku je iná interná tabuľka, je prázdny.

Používateľom definovaný kľúč môže obsahovať ľubovoľné stĺpce internej tabuľky, ktoré nie sú referenciami ani internými tabuľkami. Interné tabuľky s pouužívateľom definovaným kľúčom sa nazývajú kľúčové tabuľky. Pri definovaní kľúča tabuľky je, samozrejme, dôležité poradie.

Typ tabuľky

Typ tabuľky určuje, ako bude ABAP pristupovať k jednotlivým tabuľkovým záznamom.

Existujú tri typy interných tabuliek:

Štandardné tabuľky majú interný lineárny index. Indexy interných tabuliek sú udržiavané v stromoch. V tomto prípade náklady na udržiavanie indexu rastú logaritmicky, a nie lineárne od počtu riadkov. Systém môže pristupovať k záznamom buď použitím indexu, alebo kľúča. Doba odozvy pre prístup pomocou kľúča je lineárne závislá od počtu riadkov v tabuľke. Kľúč štandardnej internej tabuľky je vždy NON-UNIQUE. Nie je možné špecifikovať UNIQUE kľúč. To znamená, že štandardné tabuľky môžu byť naplnené veľmi rýchlo, keďže systém nemusí kontrolovať existenciu záznamov s rovnakými hodnotami kľúčových polí.

Utriedené tabuľky sú vždy uložené utriedené podľa kľúča. Tiež majú interný index. Systém môže k záznamom pristupovať pomocou indexu tabuľky alebo kľúča. Doba odozvy pri prístupe pomocou kľúča je logaritmicky závislá od počtu tabuľkových záznamov, keďže systém využíva binárne hľadanie. Kľúč utriedenej internej tabuľky môže byť UNIQUE alebo NON-UNIQUE.

Štandardné a utriedené tabuľky sa vo všeobecnosti nazývajú indexové tabuľky.

Hashované tabuľky nemajú lineárny index. Je možné k nim pristupovať jedine pomocou kľúča. V prípade prístupu pomocou iných polí ako

celého kľúča sa tabuľka správa ako štandardná. Doba odozvy pre prístup k záznamu je nezávislá od počtu tabuľkových záznamov a je konštantná, keďže systém pristupuje k tabuľkovým záznamom pomocou hashovacieho algoritmu. Kľúč hashovanej tabuľky musí byť UNIQUE.

Výber typu tabuľky

Typ tabuľky a spôsob prístupu k nej závisí na tom, ako budú typické tabuľkové operácie najčastejšie vykonávané.

Štandardné tabuľky

Štandardné tabuľky sú najvhodnejším typom v prípade prístupu k jednotlivý dátam pomocou indexu. Indexový prístup je najrýchlejší možný prístup. Tabuľky by sa mali napĺňať príkazom APPEND a čítať, modifikovať a mazať špecifikáciou indexu (INDEX dodatok s relevantným ABAP príkazom). Prístupový čas pre štandardnú tabuľku rastie lineárne s počtom tabuľkových záznamov. Ak je potrebný prístup pomocou kľúča, štandardné tabuľky sú užitočné, ak je možné naplniť a spracovať tabuľku v jednotlivých krokoch. Napríklad ak naplníme tabuľku pridávaním záznamov, potom ju utriedime, tak pri použití binárneho vyhľadávania s kľúčovým prístupom je doba odozvy logaritmicky závislá od počtu tabuľkových záznamov.

Utriedené tabuľky

Utriedené tabuľky sú najvhodnejším typom, ak potrebujeme tabuľku, ktorá je utriedená v každom kroku počas napĺňania. Táto tabuľka sa napĺňa príkazom INSERT. Záznamy sú vkladané na základe triediacej sekvencie definovanej cez kľúč tabuľky. Hocijaký neplatný záznam je rozpoznaný hneď pri pokuse vložiť ho do tabuľky. Doba odozvy pre prístup pomocou kľúča je logaritmicky závislá od počtu tabuľkových záznamov, keďže systém používa vždy binárne vyhľadávanie. Utriedené tabuľky sú užitočné pre čiastočne sekvenčné spracovanie v LOOPe, ak špecifikujeme začiatok tabuľkového kľúča vo WHERE podmienke.

Hashované tabuľky

Hashované tabuľky sú najvhodnejším typom pre ľubovoľnú tabuľku, kde je najčastejšou operáciou kľúčový prístup. K hashovaným tabuľkám nie je možné pristupovať pomocou indexu. Čas odozvy pre kľúčový prístup zostáva konštantný bez ohľadu na počet tabuľkových záznamov. Rovnako ako databázové tabuľky, tak aj hashované tabuľky majú vždy jedinečný kľúč. Hashované tabuľky sú vhodné, ak chceme skonštruovať a používať internú tabuľku, ktorá reflektuje databázovú tabuľku, alebo pre spracovanie veľkého množstva dát.[3]

Žiaľ, vo väčšine programov programátor nerozlišuje jednotlivé typy tabuliek a využíva deklaráciu tabuľky bez špecifikovania typu tabuľky a jej kľúča, keďže táto špecifikácia nie je povinná (aj keď v novších verziách je to v OO kontexte považované za zastaralý spôsob deklarácie tabuľky). V tom prípade je tabuľka zadeklarovaná ako štandardná tabuľka bez kľúča. Použitie nevhodného typu tabuľky však môže byť z hľadiska výkonnosti obrovským problémom, ako si ukážeme na príkladoch. Vo väčšine prípadov je to práve práca s internými tabuľkami, čo najviac zaťažuje aplikačný server.

Pozrime sa na konkrétne prípady použitia a obmedzenia pre jednotlivé operácie:

Základom pri práci s internými tabuľkami je ich napĺňanie záznamami. To je možne robiť naraz alebo po jednom zázname, alebo po skupine záznamov.

V prípade, že chceme vkladať záznamy po jednom a chceme tabuľku udržiavať utriedenú v každom kroku, je vhodné použitie utriedených tabuliek, do ktorých vkladáme pomocou príkazu INSERT. Svojpomocné vyhľadávanie indexu, kam má byť nový záznam vložený, je príliš pracný, avšak pri použití štandardných tabuliek je možné túto tabuľku po každom príkaze APPEND utriediť.

3 http://help.sap.com/saphelp_nw04/helpdata/EN/fc/eb35de358411d1829f0000e829fbfe/content.htm

Teda pre utriedenú tabuľku by kód vyzeral nasledovne:

1.)

```
LOOP AT gt_table INTO gs_line.
  INSERT gs_line INTO TABLE gt_sorted_table.
ENDLOOP.
```

Pri použití štandardnej tabuľky nasledovne:

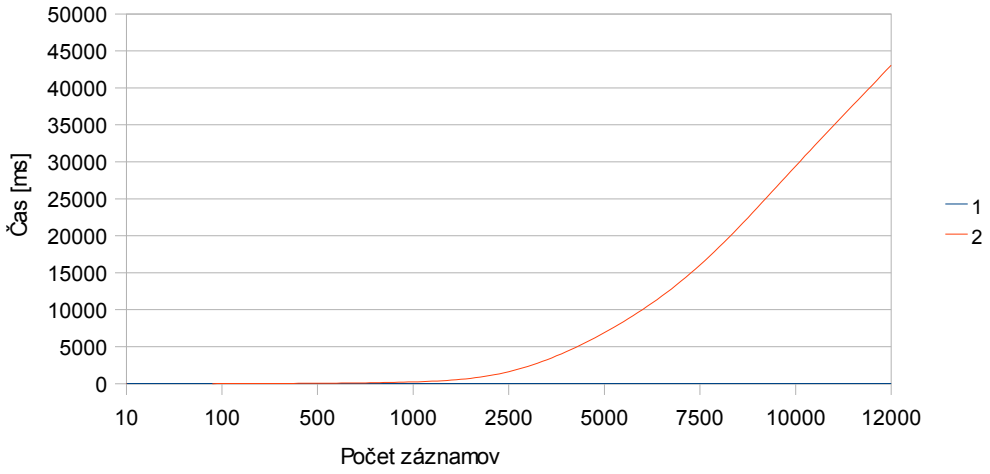
2.)

```
LOOP AT gt_table INTO gs_line.
  APPEND gs_line TO gt_sorted_table.
  SORT gt_sorted_table BY sort_field1 sort_field2...
ENDLOOP.
```

Ako vidíme, pri použití štandardnej tabuľky by sa príkaz SORT vykonával zbytočne veľa ráz, pričom jeho časová náročnosť s počtom záznamov rastie. Poďme sa pozrieť na namerané výsledky:

Počet záznamov\spôsob	1	2
10	0,02	0,07
100	0,15	2,19
500	0,86	55,46
1000	1,97	234,69
2500	5,95	1621,78
5000	13,5	6898,77
7500	21	16048,43
10000	30,24	29428,65
12000	35,8	43068,26

Tabuľka č.4 - Doby trvania behu jednotlivých spôsobov vzhľadom na počet záznamov v milisekundách.



Z nameraných výsledkov je zrejmé, že pre tabuľky, ktoré potrebujeme mať v každom kroku utriedené, sú utriedené tabuľky omnoho vhodnejšie a použitie štandardných tabuliek je absolútne nevhodné.

Pri napĺňaní tabuliek jednorázovo pomocou SELECT INTO TABLE sa v prípade, že je cieľová oblasť utriedená tabuľka, záznamy automaticky utriedia. V prípade, že má tabuľka definovaný unikátny kľúč, vznikajú ďalšie dodatočné náklady na kontrolu duplicitných záznamov. Pozrime sa, aké sú tieto náklady v skutočnosti.

Porovnával som 4 rôzne spôsoby:

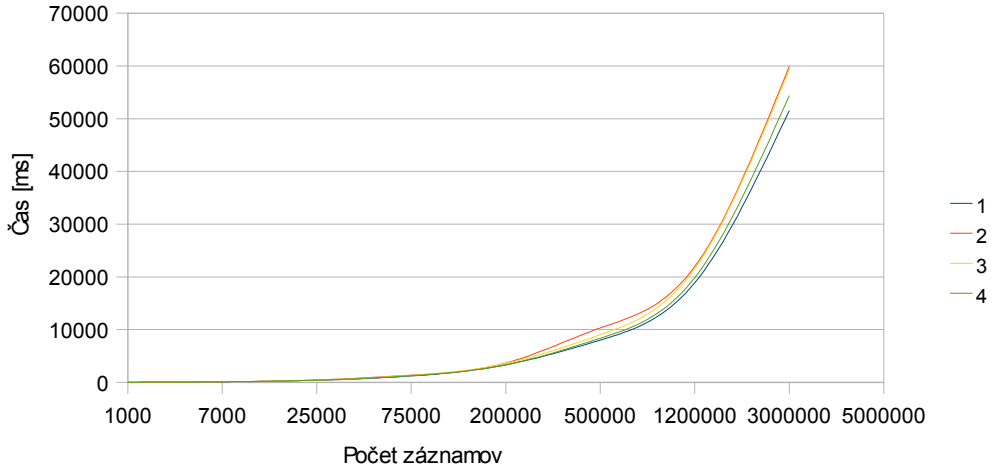
1.) selekcia dát do štandardnej tabuľky
2.) selekcia dát do utriedenej tabuľky s unikátnym kľúčom
3.) selekcia dát do utriedenej tabuľky s neunikátnym kľúčom
4.) selekcia dát do hashovanej tabuľky s unikátnym kľúčom

V prípade špecifikácie kľúča som kľúč pre internú tabuľku zvolil zhodný s databázovým kľúčom tabuľky, ktorá bola zdrojom dát pre samotný select príkaz.

Namerané výsledky boli nasledovné:

Počet záznamov\spôsob	1	2	3	4
1000	17,68	18,21	17,74	17,12
7000	119,9	126,29	125,07	121,28
25000	407,8	448,71	445,01	420,19
75000	1209,59	1357,75	1334,84	1256,67
200000	3325,14	3663,74	3574,6	3339,12
500000	8009,39	10323,56	9041,08	8361,5
1200000	18882,06	21940,23	21559,61	19855,59
3000000	51548,05	59925,52	59325,13	54364,86

Tabuľka č.5 - Doby trvania behu jednotlivých spôsobov vzhľadom na počet záznamov v milisekundách.



Ako vidno z nameraných výsledkov, dodatočné časové nároky oproti štandardným tabuľkám nie sú veľmi výrazné, a preto by nemali byť dôvodom, prečo uprednostniť štandardnú tabuľku pred utriedenou alebo hašovanou tabuľkou v prípade, že tie sú na ďalšie použitie vhodnejšie. Pri ďalších operáciách s internými tabuľkami sú totiž rozdiely z výkonnostného hľadiska oveľa priepastnejšie.

Pozrime sa napríklad na hľadanie záznamu na základe plne špecifikovaného kľúča pre jednotlivé typy tabuliek.

Porovnám 4 prístupy:

1.) čítanie zo štandardnej tabuľky

```
  DATA: gt_tab2 TYPE STANDARD TABLE OF db_tab2.
     .
     .
     .
  LOOP AT gt_tab1 ASSIGNING <ls_tab1>.
    READ TABLE gt_tab2 WITH KEY key1 = <ls_tab1>-key1
key2 = gc_key2 key3 = gc_key3 ... TRANSPORTING NO FIELDS.
  ENDLOOP.
```

2.) utriedenie štandardnej tabuľky a potom čítanie z nej pomocou binárneho vyhľadávania

```
  DATA: gt_tab2 TYPE STANDARD TABLE OF db_tab2.
     .
     .
     .
  SORT gt_tab2 BY key1 key2 ...
  LOOP AT gt_tab1 ASSIGNING <ls_tab1>.
    READ TABLE gt_tab2 WITH KEY key1 = <ls_tab1>-key1
key2 = gc_key2 key3 = gc_key3 ... BINARY SEARCH
TRANSPORTING NO FIELDS.
  ENDLOOP.
```

3.) čítanie z hashovanej tabuľky

```
  DATA: gt_tab2 TYPE HASHED TABLE OF db_tab2 WITH UNIQUE
KEY key1 key2...
     .
     .
     .
  SORT gt_tab2 BY key1 key2 ...
  LOOP AT gt_tab1 ASSIGNING <ls_tab1>.
    READ TABLE gt_tab2 WITH KEY key1 = <ls_tab1>-key1
```

```
key2 = gc_key2 key3 = gc_key3 TRANSPORTING NO FIELDS.
  ENDLOOP.
```

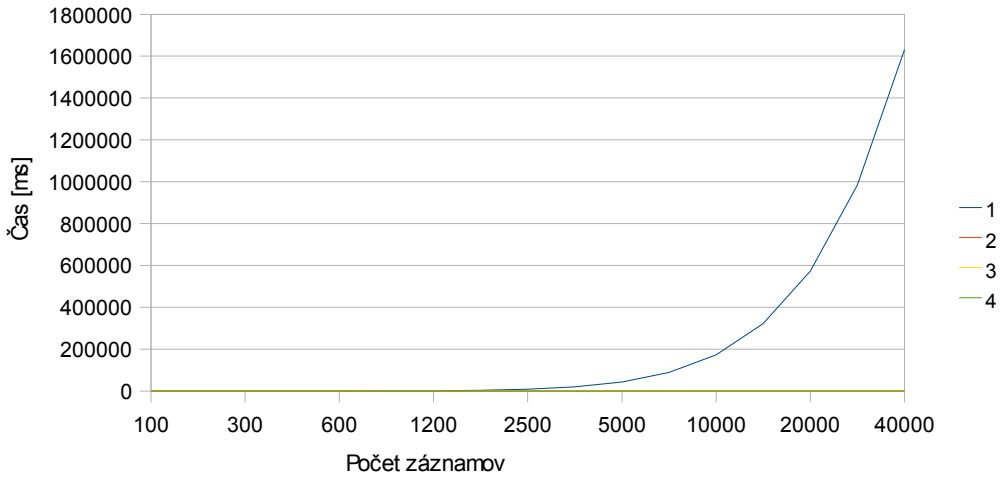
4.) čítanie z utriedenej tabuľky

```
  DATA: gt_tab2 TYPE SORTED TABLE OF db_tab2 WITH UNIQUE
KEY key1 key2...
      .
      .
      .
  SORT gt_tab2 BY key1 key2 ...
  LOOP AT gt_tab1 ASSIGNING <ls_tab1>.
    READ TABLE gt_tab2 WITH KEY key1 = <ls_tab1>-key1
key2 = gc_key2 key3 = gc_key3 TRANSPORTING NO FIELDS.
  ENDLOOP.
```

Vo všetkých príkladoch uvažujem, že kľúč tabuľky gt_tab2 je nadkľúčom tabuľky gt_tab1 (napríklad ak mám tabuľku a k nej prislúchajúcu jazykovú tabuľku). Pri príkaze READ nešpecifikujem cieľovú oblasť, ale používam TRANSPORTING NO FIELDS, pretože ma zaujíma samotná doba prístupu, a nie časová zložitosť kopírovania riadku tabuľky do cieľovej oblasti, ktorá je aj tak vo všetkých prípadoch rovnaká. Merania boli vykonávané na príkladoch textových tabuliek, kde som použil pomer riadkov gt_tab1 ku gt_tab2 ¼, pretože som vychádzal zo systémov, kde boli nainštalované štyri jazykové sady. Samozrejme, pri inom pomere by boli namerané výsledky iné. V tabuľke je v stĺpci počet záznamov uvedený počet záznamov tabuľky gt_tab1.

Počet záznamov\spôsob	1	2	3	4
100	2,73	0,38	0,11	0,16
300	25,07	1,22	0,32	0,57
600	106,51	2,61	0,63	1,02
1200	358,95	5,81	1,32	2,4
2500	580,61	12,01	2,92	5,55
5000	1857,99	25,91	6,1	12,56
10000	6348,71	59,07	14,27	28
20000	28491,77	126,42	31,58	60,68
40000	1631666,72	267,32	66,3	125,48

Tabuľka č.6 - Doby trvania behu jednotlivých spôsobov vzhľadom na počet záznamov v milisekundách.



Využitie štandardných tabuliek je na prístup pomocou kľúča bez použitia binárneho vyhľadávania absolútne nevhodný a býva častým dôvodom nepoužiteľnosti programu. Ak máme plne špecifikovaný kľúč, je jednoznačne najlepším riešením použitie hashovaných tabuliek.

Hashované tabuľky môžu výrazne prispieť k skráteniu doby behu programu v prípade, že pre dodatočné kontroly, resp. dodatočné získavanie hodnôt niektorých polí, ktoré sa bežne robia pomocou príkazu SELECT SINGLE, sa nahradia čítaním z hashovanej tabuľky, ktorá bola predtým naplnená potrebnými hodnotami.

Používanie príkazu SELECT SINGLE so špecifikovaním plného kľúča v rámci nejakej slučky (LOOP...ENDLOOP, SELECT...ENDSELECT) je v SAPe veľmi často používané, pretože doťahovanie dodatočných údajov, resp. kontrola hodnôt je často potrebná, a tento spôsob sa z výkonnostného hľadiska zdá byť optimálny, keďže prístup pomocou primárneho indexu je vo všeobecnosti rýchly oproti iným prístupom do databázy. Využitie hashovaných tabuliek si ukážeme na prípade kontroly existencie záznamu v nejakej kontrolnej tabuľke. (Načítavanie dodatočných hodnôt sa robí analogicky ako kontrola existencie záznamu, preto je zbytočné uvádzať oba príklady, a navyše pri kontrole existencie existuje aj ďalší spôsob, ktorý využíva príkaz COUNT.)

10 Kontrola existencie záznamu

Kontrola existencie záznamu sa využíva veľmi často, najmä v slučkách. Väčšinou rozhoduje o tom, či má spracúvanie pre daný záznam pokračovať alebo má byť ukončené, resp. rozhoduje o tom, ako vetvou má program pokračovať alebo sa príznak existencie nastaví iba ako atribút daného riadku. Keďže ide opäť o operáciu vykonávanú v slučke, ktorá môže byť vykonávaná obrovský počet razí, je dôležité zvoliť správny spôsob ako ju zrealizovať. Dva najčastejšie používané spôsoby sú SELECT SINGLE a SELECT … UP TO 1 ROWS.

V ABAP jazyku môžu tieto spôsoby vyzerať približne nasledovne:

1.)

```
LOOP AT gt_tab1 ASSIGNING <ls_tab1>.
  SELECT SINGLE * FROM db_tab2 INTO ls_tab2
    WHERE key1 EQ <ls_tab1>-field1
      AND key2 EQ <ls_tab1>-field2
      .
      .
      .
  IF sy-subrc EQ 0.
    lv_existence_flag = 'X'.
  ELSE.
    lv_existence_flag = space.
  ENDIF.
ENDLOOP.
```

2.)

```
LOOP AT gt_tab1 ASSIGNING <ls_tab1>.
  SELECT * FROM db_tab2 INTO ls_tab2 UP TO 1 ROWS
    WHERE key1 EQ <ls_tab1>-field1
      AND key2 EQ <ls_tab1>-field2
```

```
      .
      .
      .
    ENDSELECT.
    IF sy-subrc EQ 0.
      lv_existence_flag = 'X'.
    ELSE.
      lv_existence_flag = space.
    ENDIF.
  ENDLOOP.
```

Rozdiel v týchto dvoch spôsoboch je ten, že pri použití SELECT SINGLE databázový server vyselektuje prvý záznam spĺňajúci danú podmienku a vráti ho ako výsledok, kým pri SELECT...UP TO 1 ROWS databázový server vyselektuje všetky záznamy spĺňajúce danú podmienku a vráti prvý z nich. To síce umožní využiť napríklad agregačné funkcie a triedenie, ale na kontrolu existencie záznamu je to zbytočné. Teda pri SELECT SINGLE je očakávaná doba behu kratšia, keďže výsledok je vrátený hneď po nájdení prvého záznamu, a nemusí byť prejdená celá tabuľka. Avšak v prípade, že použijeme prístup pomocou plne špecifikovaného primárneho kľúča, nie je z hľadiska výkonnosti medzi týmito dvoma spôsobmi nejaký priepastný rozdiel, pretože množiny nájdených aj vrátených výsledkov sú rovnaké, a teda buď prázdne, alebo jednoprvkové. Tiež pri plne špecifikovanom primárnom kľúči sa nepoužije FULL TABLE SCAN, ale INDEX SCAN. Na základe výsledkov meraní ukážem, že ani jeden z týchto dvoch spôsobov však nie je na kontrolu existencie v slučke najvhodnejší.

Ďalším možným spôsobom na kontrolu existencie môže byť príkaz SELECT COUNT, ktorý vráti počet záznamov spĺňajúcich danú podmienku.

3.)

```
  LOOP AT gt_tab1 ASSIGNING <ls_tab1>.
    SELECT COUNT( * ) FROM db_tab2
      WHERE key1 EQ <ls_tab1>-field1
```

```
        AND key2 EQ <ls_tab1>-field2
        .
        .
        .
    IF sy-dbcnt GT 0.
      lv_existence_flag = 'X'.
    ELSE.
      lv_existence_flag = space.
    ENDIF.
  ENDLOOP.
```

Po príkaze SELECT COUNT sa nastaví hodnota premennej sy-dbcnt na počet vyhovujúcich záznamov. Výhodou pri tomto spôsobe je, že nemusí byť špecifikovaná žiadna cieľová oblasť, a teda nie sú prenášané ani žiadne dáta medzi databázovým a aplikačným serverom okrem informácie o počte záznamov.

Nevýhodou je, že rovnako ako pri SELECT … UP TO 1 ROWS je prechádzaná celá databáza a nie je jej prechádzanie ukončené po nájdení prvého záznamu. V prípade plne špecifikovaného primárneho kľúča to však nie je výrazne náročnejšie. Malým plusom hovoriacim v prospech tohto spôsobu je aj krajší kód, keďže nemusí byť deklarovaná žiadna nepotrebná pracovná oblasť, ktorá by bola cieľovou oblasťou pre selekciu dát.

Všetky tieto spôsoby sa líšia iba spôsobom kontroly existencie záznamu v databáze, nie však samotným počtom prístupov do databázy, keďže sa vykonávajú toľko krát, koľko krát sa vykonáva slučka. Skúsme sa pozrieť, ako zrealizovať požadovanú úlohu, pričom znížime počet prístupov do databázy. Keďže všetky tieto databázové prístupy sú vždy do tej istej tabuľky, je možné si túto databázovú tabuľku načítať do internej tabuľky a všetky kontroly existencie vykonávať na aplikačnej úrovni. Aby sme dosiahli najlepšie doby behu na aplikačnej úrovni, využijeme hashovanú tabuľku, z ktorej dokážeme čítať v lineárnom čase. V prípade, že poznáme množinu

hodnôt, pre ktoré budeme vykonávať kontroly už pred samotným LOOPom nad tabuľkou, môžeme zredukovať množstvo prenášaných dát pri napĺňaní internej hashovanej tabuľky pomocou FOR ALL ENTRIES, kde obmedzíme záznamy iba na tú množinu záznamov, s ktorými naozaj robíme. Kedy má toto obmedzenie pomocou FOR ALL ENTRIES kvôli výkonnosti význam a kedy spôsobuje ďalšie spomalenie, nie je predmetom tejto kapitoly a bude mu venovaná pozornosť v inej časti práce. V jazyku ABAP by spomínané riešenie pomocou hashovanej tabuľky vyzeralo nasledovne:

4.)

```
  TYPES: BEGIN OF ts_tab2_key, " structure type with key
fields of a control table
           key1 TYPE db_tab2-key1,
           key2 TYPE db_tab2-key2,
             .
             .
             .
         END OF ts_tab2_key.
  DATA:  lt_tab2 TYPE HASHED TABLE OF ts_tab2_key WITH
UNIQUE KEY key1 key2 ... " internal hashed table which
serves as a control table

* if an outer table is initial I will not continue
  CHECK NOT gt_tab1 IS INITIAL.
* i will fill the hashed table
  SELECT key1
         key2
           .
           .
           .
    FROM db_tab2 INTO TABLE lt_tab2
    FOR ALL ENTRIES IN gt_tab1 "Only those entries which
are in an outer table are necessary
    WHERE key1 EQ gt_tab1-field1
```

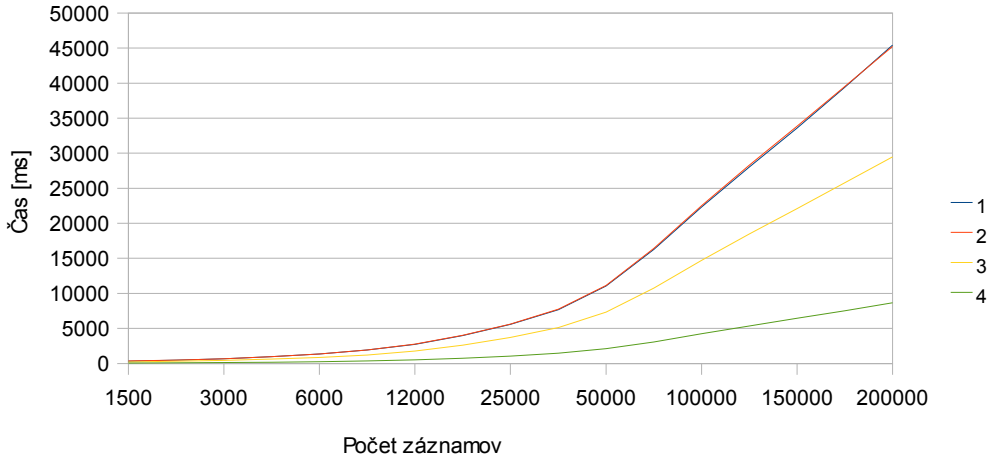
```
      AND key2 EQ gt_tab1-field2
          .
          .
          .
    .
* main LOOP
  LOOP AT gt_tab1 ASSIGNING <ls_tab1>.
* i will check entry existence in the control table
    READ TABLE lt_tab2 WITH KEY key1 = <ls_tab1>-field1
key2 = <ls_tab1>-field2 TRANSPORTING NO FIELDS.
    IF sy-subrc EQ 0.
      lv_existence_flag = 'X'.
    ELSE.
      lv_existence_flag = space.
    ENDIF.
  ENDLOOP.
```

V tomto prípade sa zredukoval počet prístupov do databázy ku kontrolnej tabuľke na jeden prístup namiesto počtu prístupov rovnému počtu riadkov tabuľky, nad ktorou je vykonávaný LOOP.

Pri meraní som získal tieto hodnoty:

Počet záznamov\spôsob	1	2	3	4
1500	343,34	346,79	220,66	65,86
3000	682,34	689,76	438,81	127,93
6000	1359,32	1369,3	872,89	255,38
12000	2731,56	2755,67	1757,44	518,12
25000	5570,39	5629,13	3691,91	1058,3
50000	11047,17	11148,19	7336,16	2095,03
100000	22312,93	22524,56	14711,8	4240,25
150000	33608,54	33870,97	22114,44	6437,99
200000	45424	45259,55	29478,31	8652,64

Tabuľka č.7 - Doby trvania behu jednotlivých spôsobov vzhľadom na počet záznamov v milisekundách.



Ako vidieť z nameraných výsledkov, využitie hashovanej tabuľky ako kontrolnej tabuľky prinieslo očakávané zlepšenie oproti ostatným spôsobom. Keďže som pri meraniach využíval plne špecifikovaný primárny kľúč, rozdiel medzi SELECT SINGLE a SELECT UP TO 1 ROWS je úplne zanedbateľný, a spôsob využívajúci SELECT COUNT je rýchlejší napriek tomu, že tiež SELECT COUNT vo všeobecnosti nekončí pri nájdení prvého záznamu, ale prechádza celú tabuľku. Získané zlepšenie je spôsobené tým, že nie sú prenášané žiadne zbytočné dáta medzi databázovým a aplikačným serverom. Samozrejme, dá sa zredukovať aj množstvo prenášaných dát pri prvých dvoch spôsoboch, ale porovnal som najbežnejšie používané spôsoby tak, ako sú používané. Pri zredukovaní prenášaných dát pre prvé dva spôsoby sa ich časová náročnosť blíži k časovej náročnosti tretieho spôsobu.

Čo sa týka najefektívnejšieho spôsobu a teda využitia hašovanej tabuľky, tu treba v prípade selekcie dát iným spôsobom dať pozor na duplikátne záznamy, ktoré by mohli spôsobiť krátky dump a použiť príkaz DISTINCT, resp. iným spôsobom zabezpečiť jedinečnosť záznamov. To, kedy má v tomto spôsobe FOR ALL ENTRIES opodstatnenie, na to sa pozriem v ďalšej kapitole.

11 FOR ALL ENTRIES

Ak sa pred príkazom WHERE nachádza jazykový prvok FOR ALL ENTRIES, komponenty štruktúry internej tabuľky môžu byť použité v SQL podmienke ako operátory porovnania s relačnými operátormi. Môžu byť použité iba vo WHERE podmienke. Špecifikované komponenty musia byť kompatibilné so stĺpcami, s ktorými sa porovnávajú. Interná tabuľka môže mať štruktúrovaný alebo elementárny typ riadka. Pre elementárny typ riadka, pre komponent musí byť špecifikovaný pseudo komponent table_line.

Celý logický výraz je vyhodnotený pre každý riadok internej tabuľky. Množina výsledkov je zjednotením množiny výsledkov z jednotlivých vyhodnotení. Riadky, ktoré sa vyskytujú duplicitne, sú automaticky odstránené z množiny výsledkov. Ak je interná tabuľka prázdna, celá WHERE podmienka je ignorovaná a všetky riadky z databázy sú vrátené ako množina výsledkov.

Logický výraz WHERE podmienky môže pozostávať z niekoľkých logických výrazov používajúcich AND a OR. Ak je však špecifikovaný prvok FOR ALL ENTRIES, musí existovať aspoň jedno porovnanie so stĺpcom internej tabuľky, ktoré môže byť špecifikované staticky alebo dynamicky. V príkaze SELECT s FOR ALL ENTRIES môže byť dodatok ORDER BY použitý iba spolu s PRIMARY KEY, teda nie je možné použiť iné triedenie ako pomocou primárneho kľúča.[4]

FOR ALL ENTRIES nám umožňuje spájať dva rôzne typy tabuliek – interné a databázové. To môže mať praktické využitie, keď dáta internej tabuľky sú získané nejakým komplikovanejším spôsobom, ako je SELECT z databázy, to znamená, že sa nedá tento príkaz nahradiť obyčajným JOINom databázových tabuliek. JOIN databázových tabuliek je z výkonnostného hľadiska vo všeobecnosti efektívnejší. Ďalším praktickým využitím môže byť nahradenie subquery. To umožní obmedziť vybrané záznamy už na databázovej úrovni namiesto aplikačnej úrovne.

4 http://help.sap.com/abapdocu/en/ABENWHERE_LOGEXP_ITAB.htm

Nevýhody a riziká FOR ALL ENTRIES

Použitie FOR ALL ENTRIES môže viesť k rôznym rizikám, respektíve k zníženiu výkonnosti, na ktoré treba pri jeho použití dbať.

Medzi základné nevýhody patria:

1.) Výmaz duplicitných riadkov v riadiacej tabuľke. Každý SELECT s dodatkom FOR ALL ENTRIES sa správa automaticky, ako keby mal špecifikovaný aj dodatok DISTINCT. To znamená, že pri jeho použití strácame informáciu o počte výskytov jednotlivých záznamov. Počet vrátených záznamov sa nemusí rovnať počtu vyhovujúcich záznamov v databáze.

2.) Ak je riadiaca interná tabuľka prázdna, WHERE podmienka je ignorovaná a všetky riadky z databázy sú vrátené ako množina výsledkov. To znamená, že pred každým vykonávaním SELECT príkazu s dodatkom FOR ALL ENTRIES by sa malo kontrolovať, či nie je tabuľka prázdna a v prípade, že je prázdna, nemal by sa samotný SELCT vykonať, pretože väčšinou sa v prípade prázdnej riadiacej tabuľky očakáva ako výsledok prázdna množina výsledkov (výnimkou môže byť použitie riadiacej internej tabuľky namiesto RANGES a SELECT-OPTIONS). Teda okrem nesprávnej logiky chodu programu, pri veľkých tabuľkách môže byť v takomto prípade množinou výsledkov obrovské množstvo dát. Ich získanie môže byť časovo veľmi náročné a môže viesť k vyťaženiu databázového servera, aplikačného servera, siete, zaplnenie redo logov atď.

3.) Dodatky PACKAGE SIZE a UP TO n ROWS sa nepredávajú databáze, ale toto obmedzenie je vykonané na aplikačnej úrovni nad množinou výsledkov, až keď je získaná z databázy celá množina výsledkov. To znamená, že tieto dodatky neznížia množstvo prenášaných dát medzi databázovým a aplikačným serverom.

4.) Vykonávanie príkazu FOR ALL ENTRIES je závislé od mnohých faktorov,

ako sú typ databázovej platformy, verzia databázovej platformy, nastavenie parametrov profilu, jadro aplikačného servra.[5] To znamená, že programátor nemá presnú predstavu o dobe behu SELECTu využívajúceho FOR ALL ENTRIES, pretože tá sa môže rapídne zmeniť so zmenou databázového servera, resp. parametrov profilu.

5.) Pri veľkých interných tabuľkách môže prísť k výkonnostným problémom, keďže FOR ALL ENTRIES musí byť transformované do komplikovaného SQL dotazu, ktorý databázový server nedokáže efektívne vykonať.

Možnosti zlepšenia výkonnosti

1. Utriedenie riadiacej tabuľky a výmaz duplikátov

Najjednoduchšou možnosťou zlepšenia výkonnosti príkazu využívajúceho FOR ALL ENTRIES je utriedenie internej tabuľky a výmaz duplikátov z nej. Výmaz duplikátov môže značným spôsobom zlepšiť čas vykonávania dotazu, pretože každý záznam internej tabuľky má vygenerovanú vlastnú časť príkazu v koncovom DB príkaze; to znamená, že zredukovanie počtu riadkov koncový dotaz zjednoduší. Pre prípad, že počet záznamov klesne odstránením duplikátov o viac ako je blokovací faktor, zníži sa aj počet vykonaných príkazov v DB. Pod pojmom duplikáty myslím v tomto prípade nie riadky, ktoré sú rovnaké, ale riadky, ktoré majú rovnakú hodnotu polí vstupujúcich do porovnania vo WHERE podmienke. To znamená, že v prípade, že potrebujeme zachovať obsah tabuľky, je vhodné vytvoriť novú internú tabuľku, ktorá bude mať ako kľúčové polia tie polia, ktoré vstupujú do porovnania vo WHERE podmienke, a žiadne iné polia obsahovať nebude.

2. Využitie parametrov

Open SQL príkaz "SELECT ... FOR ALL ENTRIES ..." je ABAP-špecifické rozšírenie SQL štandardu. Keďže v SQL štandarde nie sú žiadne podobné príkazy, tento príkaz musí byť databázovým rozhraním ABAP

5 https://service.sap.com/sap/support/notes/652634

prostredia namapovaný do jedného alebo viacerých sémanticky ekvivalentných SELECT príkazov, ktoré môžu byť spracované databázovou platformou. Niekoľko parametrov profilu umožňuje definovať, ako má databázové rozhranie vykonávať toto mapovanie s ohľadom na databázu. Konkrétne ide o nasledovné parametre:

– rsdb/prefer_join (od verzie 7.00) – ak je nastavený tento parameter na hodnotu 1, SELECT … FOR ALL ENTRIES je implementované použitím joinu (pre platformu DB6 a MS SQL od verzie SAPu 7.00, pre Oracle od verzie 7.10). Jednotlivé hodnoty sú sumarizované v subselekte.

```
SELECT * FROM <TABLE>,
(SELECT k0=<VAR1>
UNION ALL SELECT <VAR2>
UNION ALL SELECT <VAR3> ...) as q
WHERE <FIELD> = q.k0
```

– rsdb/prefer_union_all – ak je nastavený na 1, generuje linkovanie všetkých príkazov pomocou UNIONu, ak je nastavený na 0, jednotlivé podmienky sú spojené cez OR. Tento parameter má efekt, iba ak je rsdb/prefer_join nastavený na 0, ináč sa využíva JOIN. Teda na základe hodnoty tohto parametra pretransformovaný príkaz vyzerá pre hodnotu 1 nasledovne:

```
SELECT ... WHERE f = itab[1]-f
UNION ALL SELECT ... WHERE f = itab[2]-f
....
UNION ALL SELECT ... WHERE f = itab[N]-f
```

a pre hodnotu 0:

```
SELECT ... WHERE f = itab[1]-f
OR f = itab[2]-f
...
```

```
OR f = itab[N]-f
```

– rsdb/prefer_in_itab_opt – ak je nastavený na 1, príkaz, kde iba jedno pole vo WHERE podmienke záleží od internej tabuľky, je prekladaný ako príkaz využívajúci IN podmienku. Avšak to je možné len v prípade, že toto pole sa vyskytuje v príkaze ako nenegovaná podmienka rovnosti.

  ```
  SELECT ... FOR ALL ENTRIES IN itab WHERE f = itab-f.
  ```

  je preložený ako

  ```
  SELECT ... WHERE f IN (itab[1]-f, itab[2]-f, ...,
       itab[N]-f)
  ```

– rsdb/max_blocking_factor – tento parameter určuje hornú hranicu pre počet záznamov v internej tabuľke, ktoré majú byť spracované v jednom príkaze. V prípade, že počet záznamov v internej tabuľke prekročí túto hranicu, príkaz je rozdelený do viacerých príkazov pre databázu, ktorých výsledky sú potom zozbierané databázovým rozhraním a vrátené ako celkový výsledok do ABAP programu. Tento parameter nemá efekt na mapovanie pomocou IN operátora.
– rsdb/max_in_blocking_factor – tento parameter podobne ako parameter rsdb/max_blocking_factor učuje hornú hranicu pre počet záznamov internej tabuľky, ak je konkrétný príkaz mapovaný pomocou IN operátora.
– rsdb/prefer_fix_blocking – ak počet záznamov internej tabuľky nie je deliteľný max_blocking_factor, menej záznamov (resp. podmienok) je generovaných pre posledný príkaz vygenerovaný pre FOR ALL ENTRIES príkaz. Výsledkom je nový príkaz. Ak je rovnaký FOR ALL ENTRIES príkaz vykonávaný veľmi často s rôznym počtom záznamov vo vstupnej internej tabuľke, rozdielne príkazy sú generované. Ak je tento parameter nastavený na 1, sú generované najviac dva príkazy rôznej dĺžky. To je dosiahnuté opakovaním poslednej hodnoty vo vstupnej tabuľke, ako keby bola implicitne naplnená až po hodnotu blokovacieho faktoru (interná tabuľka nie je modifikovaná).

– rsdb/min_blocking_factor – ak je tento parameter nastavený na hodnotu väčšiu ako 0 a rsdb/prefer_fix_blocking je nastavený na 1, dve rôzne blokovacie faktory sú použité: max_blocking_factor a min_blocking_factor. Avšak min_blocking_factor sa používa, iba ak je počet záznamov v internej tabuľke malý - trochu zjednodušené pravidlo je, že keď je menší ako polovica hodnoty max_blocking_factor.
– rsdb/min_in_blocking_factor – analógia k rsdb/min_blocking_factor pre využitie IN operátora

Tieto parametre majú veľký dopad na celý systém a jeho výkonnosť, preto ich zmena vyžaduje dôkladnú analýzu. Napriek tomu, že by zmena týchto parametrov mohla vyriešiť lokálny problém, mohla by spôsobiť oveľa väčší globálny problém, spôsobila by totiž zmenu spôsobu vykonávania obrovského množstva dotazov. Od verzie 4.6B má programátor možnosť nastaviť tieto parametre pre individuálny SELECT využitím databázových hintov. Hinty pre FOR ALL ENTRIES by mali byť rovnako ako aj všetky ostatné hinty využívané po dôkladnom uvážení.[6]

Na základe skúseností firmy SAP s výkonnosťou u väčšiny reportov v jazyku ABAP je z výkonnostného hľadiska najlepšie využitie JOINov, potom ORov a nakoniec UNIONov[7] (porovnanie nezahŕňa využitie IN operátora, ktoré je možné využiť len za už popísaných podmienok).

Hranice výhodnosti použitia FOR ALL ENTRIES

Niekedy, keď FOR ALL ENTRIES využívame na obmedzenie počtu prenášaných záznamov z DB tabuľky za účelom skrátenia doby behu, časové náklady na vykonanie dotazu na úrovni databázy sa zvýšia natoľko, že doba behu je celkovo predĺžená. Jedným z najdôležitejších faktorov, ktoré ovplyvňujú pomer časov trvania medzi selektom bez obmedzenia a selektom s dodatkom FOR ALL ENTRIES, je pomer počtu záznamov v DB tabuľke a počtu záznamov v internej tabuľke. Ďalšími faktormi sú napríklad spôsob

6 https://service.sap.com/sap/support/notes/48230
7 https://service.sap.com/sap/support/notes/652634

vykonávania na databázovej úrovni a využitie indexových polí.

Vykonal som viacero meraní na nájdenie hranice, kedy už použitie FOR ALL ENTRIES nie je z výkonnostného hľadiska výhodné. V jednotlivých prípadoch bola táto hranica rôzna, ale pozrime sa na prípad, keď je tento pomer čo najlepší v prospech FOR ALL ENTRIES. To je prípad, keď pretransformovaný FOR ALL ENTRIES príkaz na úrovni DB má čo najjednoduchšiu formu a využíva index. To je prípad, keď sa dotaz transformuje na dotaz využívajúci IN operátor. Preto použijem obmedzenie iba na jedno pole internej tabuľky a toto pole bude v DB tabuľke prvým poľom kľúča, aby som umožnil čo najefektívnejšie využitie indexového vyhľadávania. Samozrejme, internú tabuľku budem mať utriedenú a bez duplikátov.

Porovnávam teda nasledovné dotazy:

1.)

```
SELECT field1 INTO TABLE lt_tab1 FROM db_tab
   FOR ALL ENTRIES IN lt_for_all_entries_tab
    WHERE key1 EQ lt_for_all_entries_tab-key1
  .
```

a

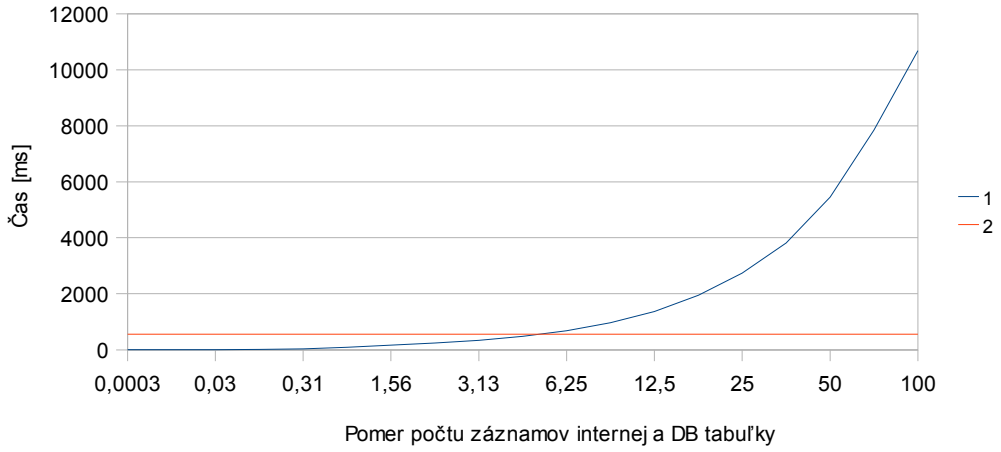
2.)

```
SELECT field1 INTO TABLE lt_tab1 FROM db_tab.
```

Doby trvania vykonania jednotlivých dotazov boli v závislosti od pomeru počtu záznamov internej a databázovej tabuľky nasledovné.

Pomer počtu záznamov(v%) \ spôsob	1	2
0,0003	0,25	560
0,03	3,94	560
0,31	34,24	560
1,56	169	560
3,13	338,98	560
6,25	683,26	560
12,5	1369,38	560
25	2739,4	560
50	5450,65	560
100	10681,78	560

Tabuľka č.8 - Doby trvania behu jednotlivých spôsobov vzhľadom na pomer počtu záznamov internej a databázovej tabuľky v milisekundách.



V tomto konkrétnom prípade sa hranica výhodnosti použitia FOR ALL ENTRIES vzhľadom na behu dobu nachádza na úrovni 5,3 %. Vo zvyšných testovaných prípadoch, kde som použil komplikovanejšie FOR ALL ENTRIES podmienky neumožňujúce DB dotazu na úrovni databázy využívať IN operátor ani index, sa táto hranica často znížila pod úroveň 1 %. Z toho vyplýva, že ak chceme použiť FOR ALL ENTRIES v rámci optimalizácie doby behu a nie ako nutnosť, musíme dbať na očakávaný pomer záznamov v internej a DB tabuľke. Tiež musíme zvážiť, či sa na DB úrovni využije index. Samozrejme pri selekcii všetkých záznamov sa zvyšujú pamäťové nároky, preto je skrátenie doby behu sprevádzané plytvaním pamäťových zdrojov a treba zvážiť ktorý z týchto dvoch faktorov je kritický.

12 Zákaznícky vývoj v rôznych moduloch

Každý modul v SAPe obsahuje nejaké tabuľky, ktoré môžeme považovať za kľúčové tabuľky pre daný modul. Pri zákazníckom vývoji sa stáva, že zákaznícke programy a rozšírenia, ktoré pristupujú do týchto tabuliek, majú zlú výkonnosť. Najväčším problémom často bývajú práve prístupy k týmto tabuľkám.

Dôvodom je, že SAP neobsahuje vo veľkom množstve základných tabuliek sekundárne indexy v rámci štandardnej dodávky. Namiesto toho obsahuje proprietárne indexové tabuľky, ktoré umožňujú efektívny prístup. Aby bolo možné ich použiť efektívne, vyžaduje sa základná znalosť dátového modelu SAP. Spomeniem najznámejšie chyby, ktoré sa robia a možné alternatívy. Predpokladom je, že indexy sú v systéme korektne spravované. To je zvyčajne automatické. Príklady pre niektoré moduly:

Modul SD (Pozri OSS Note 185530)

1.) Prístupy k predajným dokladom (zákazkám) (tabuľky VBAK, VBAP)

a.) Vyhľadávanie zákaziek podľa čísla zákazníka (pole VBAK-KUNNR)

Nesprávne:

```
SELECT FROM vbak WHERE kunnr = ...
```

Správne:

```
SELECT FROM vakpa WHERE kunde = ...
SELECT FROM vbak WHERE vbeln = vakpa-vbeln.
```

b.) Vyhľadávanie položiek zákaziek podľa čísla materiálu (pole VBAP-MATNR):

Nesprávne:

```
SELECT FROM vbap WHERE matnr = ...
```

Správne:

```
SELECT FROM vapma WHERE matnr = ...
SELECT FROM vbap WHERE vbeln = vapma-vbeln
AND posnr = vapma-posnr
```

2.) Prístupy k odbytovým dokladom (dodávkam) (tabuľky LIKP, LIPS)

a.) Vyhľadávanie dodávok podľa čísla zákazníka (pole LIKP-KUNNR):

Nesprávne:

```
SELECT FROM likp WHERE kunnr = ...
```

Správne:

```
SELECT FROM vlkpa WHERE kunde = ...
SELECT FROM likp WHERE vbeln = vlkpa-vbeln.
```

b) Vyhľadávanie položiek dodávky podľa čísla materiálu (pole LIKP-MATNR):

Nesprávne:

```
SELECT FROM lips WHERE matnr = ...
```

Správne:

```
SELECT FROM vlpma WHERE matnr = ...
SELECT FROM lips WHERE vbeln = vlpma-vbeln
AND posnr = vlpma-posnr
```

c) Vyhľadávanie dodávok podľa zákazky (referenčný doklad, LIPS-VGBEL):

Nesprávne:

```
SELECT FROM lips WHERE vgbel = ...
```

Správne:

```
SELECT FROM vbfa WHERE VBELV = ... and VBTYP_N = 'J'
SELECT FROM lips WHERE vbeln = vbfa-vbeln
AND posnr = vbfa-posnn
```

3.) Prístupy k faktúram (tabuľky VBRK, VBRP)

a.) Vyhľadávanie faktúr podľa čísla zákazníka („platcu“) (pole VBRK-KUNRG):

Nesprávne:

```
SELECT FROM vbrk WHERE kunrg = ...
```

Správne:

```
SELECT FROM vrkpa WHERE kunde = ...
SELECT FROM vbrk WHERE vbeln = vrkpa-vbeln
```

b.) Vyhľadávanie položiek faktúry podľa čísla materiálu (pole VBRP-MATNR):

Nesprávne:

```
SELECT FROM vbrp WHERE matnr = ...
```

Správne:

```
SELECT FROM vrpma WHERE matnr = ...
SELECT FROM vbrp WHERE vbeln = vrpma-vbeln
AND posnr = vrpma-posnr
```

c.) Vyhľadávanie faktúr podľa čísla dodávky (referenčný doklad,pole VBRP-VGBEL):

Nesprávne:

```
SELECT FROM vbrp WHERE vgbel = ...
```

Správne:

```
SELECT FROM vbfa WHERE vbtyp_n = 'M'
AND vbelv = ...
SELECT FROM vbrp WHERE vbeln = vbfa-vbeln
AND posnr = vbfa-posnn
```

d) Vyhľadávanie faktúr podľa čísla zákazky (predchádzajúci doklad, pole VBRP-AUBEL):

Nesprávne:

```
SELECT FROM vbrp WHERE aubel = ...
```

Správne:

```
SELECT FROM vbfa WHERE vbtyp_n = 'M'
AND vbelv = ...
SELECT FROM vbrp WHERE vbeln = vbfa-vbeln
AND posnr = vbfa-posnn
```

4.) Iné prístupy v module SD:

a.) Tok dokladov:

Nesprávne:

```
SELECT vbelv FROM vbfa WHERE vbeln ...
```

V tabuľke VBFA je použitý iba referenčný doklad na vyhľadávanie referujúceho dokladu (napríklad dodávka pre zákazku). Vyhľadávanie

opačným smerom nedáva v tejto tabuľke zmysel, keďže referenčné doklady (napríklad zákazka pre dodávku) sú uložené priamo v tabuľkách týchto dokladov. Preto čítanie z tabuľky VBFA je jednosmernou cestou.

Správne:

```
SELECT vgbel FROM lips WHERE vbeln = ...; or
SELECT vgbel FROM vbrp WHERE vbeln = ...; or
SELECT aubel FROM vbrp WHERE vbeln = ...
```

b.) Hľadanie položky manipulačnej jednotky (balenia) pre dodávku:

Nesprávne:

```
SELECT FROM vepo WHERE vbtyp = 'J'
AND vbeln = i_lips-vbeln
```

Správne:

```
SELECT FROM vbfa WHERE vbtyp_n = 'X'
AND vbelv = i_lips-vbeln
SELECT FROM vepo WHERE venum = vbfa-vbeln
```
[8]

Modul PP a PM (pozri OSS Note 187906)

1.) Prístup k spätným hláseniam zákazky (tabuľka AFRU)

a.) Vyhľadávanie spätných hlásení k zákazkám (pole AFRU-AUFNR):

Nesprávne:

```
SELECT FROM afru WHERE aufnr = ...
```

Správne:

```
SELECT aufpl FROM afko WHERE aufnr = <afru-aufnr>
SELECT rueck FROM afvc WHERE aufpl = <afko-aufpl>
SELECT ..... FROM afru WHERE rueck = <afvc-rueck>
```

Varovanie: Z tabuľky AFVC sa vráti ako výsledok jedna hodnota pre operáciu pre jeden "RUECK".

2.) Prístup k PPS zákazkám (tabuľky AFKO, CAUFV, AUFK)

a.) Hľadanie zákaziek pre rezerváciu (pole AFKO-RSNUM; CAUFV-RSNUM):

Nesprávne:

8 https://service.sap.com/sap/support/notes/185530

```
SELECT FROM afko WHERE rsnum = ...
```

Správne:

```
SELECT aufnr FROM resb WHERE rsnum = <afko-rsnum>
SELECT ..... FROM afko WHERE aufnr = <resb-aufnr>
```

Poznámka: Rovnaká prístupová cesta sa vzťahuje na CAUFV a AUFK.

b.) Vyhľadávanie pre odbytové zákazky(tabuľky VBAP-VBELN, VBAP-POSNR; VBFA-VBELN, VBFA-POSNN atď.)

Nesprávne:

```
SELECT FROM AUFK WHERE KDAUF = <VBAP-VBELN>
AND KDPOS = <VBAP-POSNR>
```

Správne:

```
SELECT FROM AFPO WHERE KDAUF = <VBAP-VBELN>
AND KDPOS = <VBAP-POSNR>
SELECT FROM AUFK WHERE AUFNR = AFPO-AUFNR.
```

Poznámka: Tabuľky AFKO a CAUFV môžu byť selektované analogicky k AUFK používajúc číslo zákazky AUFNR.

3.) Prístup k rezerváciám a sekundárnym potrebám (tabuľky: RESB, MDRS ATP_RESB)

a.) Vyhľadávanie rezervácií pre PPS zákazky(pole RESB-AUFNR):

Nesprávne:

```
SELECT FROM resb WHERE AUFNR = ...
```

Správne:

```
SELECT rsnum FROM afko WHERE aufnr = <resb-aufnr>
SELECT ..... FROM resb WHERE rsnum = <afko-rsnum>
```

Poznámka:

Rovnaká prístupová cesta sa vzťahuje na MDRS a ATP_RESB.

b.) Vyhľadávanie rezervácií pre plánované zákazky (pole RESB-PLNUM):

Nesprávne:

```
SELECT FROM resb WHERE PLNUM = ...
```

Správne:

```
SELECT rsnum FROM plaf WHERE plnum = <resb-plnum>
SELECT ..... FROM resb WHERE rsnum = <plaf-rsnum>
```

Poznámka: Rovnaká prístupová cesta sa vzťahuje na MDRS a ATP_RESB.[9]

Modul MM/WM (pozri OSS Note 191492)

1.) Prístup k skladovým príkazom

a.) cez číslo jednotky skladu

Nesprávne:

```
SELECT FROM LTAP WHERE VLENR = ....
```

alebo

```
SELECT FROM LTAP WHERE NLENR = ...
```

Správne:

```
SELECT FROM LEIN WHERE LENUM = ...
SELECT FROM LTAP WHERE LGNUM = LEIN-LGNUM and
AND TANUM = LEIN-BTANR
AND TAPOS = LEIN-BTAPS.
```

b.) cez SD dodávku

Nesprávne:

```
SELECT FROM LTAP WHERE NLPLA = LIPS-VBELN
AND POSNR = LIPS-POSNR.
```

Správne:

```
SELECT FROM VBFA WHERE VBELV = LIPS-VBELN
AND POSNV = LIPS- POSNR
AND VBTYP_N = 'Q'.
SELECT FROM LTAP WHERE LGNUM = LIPS-LGNUM
AND TANUM = VBFA-VBELN
AND TAPOS = VBFA-POSNN.
```

2.) Prístup k materiálovým dokladom

a.) cez číslo nákupného dokladu

Nesprávne:

```
SELECT FROM MSEG WHERE EBELN = ...
and EBELP = ...
```

9 https://service.sap.com/sap/support/notes/187906

Správne:

```
SELECT FROM EKBE WHERE EBELN = ..
AND EBELP = ...
AND VGABE IN (1,6,7,8,9).
SELECT FROM MSEG WHERE MBLNR = EKBE-BELNR
AND MJAHR = EKBE-GJAHR
AND ZEILE = EKBE-BUZEI.
```

b.) cez číslo skladového príkazu

Nesprávne:

```
SELECT FROM MSEG WHERE TANUM = ...
```

Správne:

```
SELECT FROM LTAP WHERE TANUM = ...
SELECT FROM MSEG WHERE MBELN = LTAP-WENUM
AND MJAHR = <požadovaný rok>
AND ZEILE = LTAP-WEPOS.
```

c.) cez číslo skladovej potreby

Nesprávne:

```
SELECT MSEG WHERE TBNUM = ...
```

Správne:

```
SELECT LTBK WHERE TBNUM = ....
SELECT MSEG WHERE MBLNR = LTBK-MBLNR
AND MJAHR = LTBK-MJAHR.
```

d.) cez číslo dodávateľa

Nesprávne:

```
SELECT FROM MSEG WHERE LIFNR = ...
```

Správne:

```
SELECT EKKO WHERE LIFNR = ....
SELECT EKBE WHERE EBELN = EKKO-EBELN
AND VGABE = '1'.
SELECT MSEG WHERE MBLNR = EKBE-BELNR
AND MJAHR = EKBE-GJAHR
AND ZEILE = EKBE-BUZEI.
```

3.) Prístup k požiadavkám na objednávku

a.) cez číslo rezervácie

Nesprávne:

```
SELECT FROM EBAN WHERE EBELN = ....
AND EBELP = ....
```

Správne:

```
SELECT FROM EKET WHERE EBELN = ....
EBELP = ....
SELET FROM EBAN WHERE BANFN = EKET-BANFN
AND BANFPO = EKET-BANFPO.
```

4.) Prístup k došlým faktúram

a.) cez číslo nákupného dokladu

Nesprávne:

```
SELECT FROM RSEG WHERE EBELN = ...
and EBELP = ...
```

Správne:

```
SELECT FROM EKBE WHERE EBELN = ...
AND EBELP = ...
AND VGABE IN (2,3,P).
SELECT FROM RSEG WHERE BELNR = EKBE-BELNR
AND GJAHR = EKBE-GJAHR
AND BUZEI = EKBE-BUZEI.
```

Poznámka: Špecifikovaním typu transakcie VGABE sú hodnoty obmedzené na relevantné materiálové pohyby. S GJAHR a BUZEI je primárny kľúč plne špecifikovaný.[10]

V mnohých uvedených príkladoch boli využité prístupy k viacerým tabuľkám. Tie je možné nahradiť vytvorením view-u.

Uvedené príklady sú len z niektorých modulov a popisujú len základné prístupy. Pri vývoji programu pristupujúceho k nejakej základnej databázovej tabuľke s kmeňovými dátami je potrebné zistiť, či k nej neexistuje indexová tabuľka.

10 https://service.sap.com/sap/support/notes/191492

13 Open SQL hinty

Open SQL príkazy prechádzajú cez databázový optimalizátor, ktorý má za úlohu vygenerovať plán, podľa ktorého bude SQL dotaz vykonaný. Plán vykonávania dotazu má veľký dopad na dobu jeho trvania. Existuje RBO (rule based optimizer) a CBO (cost based optimizer). RBO vytvára plán na základe pravidiel a CBO na základe štatistík. Môže sa stať, že tvorca programu má lepší prehľad o údajoch v databáze ako optimalizátor, a vtedy môže použiť od verzie SAP-u 4.5 tzv. hinty a tým zlepšiť dobu trvania vyhodnocovania dotazov. Hintami určuje optimalizátoru spôsob prístupu optimalizátora k optimalizácii (v prípade, ak podporuje aj RBO aj CBO), akú prístupovú cestu má optimalizátor zvoliť, poradie a spôsob spájania dvoch tabuliek, prípadne transformáciu dotazu. Použiteľné hinty sú, samozrejme, závislé od databázovej platformy a jej verzie.

Vo všeobecnosti platí:

- Open SQL hinty sú vyhodnocované iba pre transparetné tabuľky a viewy nad nimi. Pre príkazy na databázový buffer a pre poolové a clustrové tabuľky nie sú vyhodnocované.
- Open SQL hinty by sa mali využívať v zmysle korekcie, ak neexistuje iné riešenie.
- Hinty môžu byť vyhodnotené iba počas runtime a nemôžu byť skontrolované kontrolou syntaxe pre ABAP. Nesprávne hinty môžu spôsobiť runtime chybu alebo sa môžu ignorovať v závislosti od databázy.
- Hinty sú vždy špecifické práve pre jeden release jedného databázového systému.
- Databázový Open SQL hint sa viaže pre jeden Open SQL príkaz a ovplyvňuje iba ten. V tomto prípade nemá žiadny funkčný efekt. To znamená, že hoci ovplyvňuje trvanie procesu, neovplyvňuje množinu výsledkov príkazu.

Open SQL notácia

Hinty sa v Open SQL uvádzajú pomocou klauzuly %_HINTS, ktorá je v poradí poslednou klauzulou SELECT, DELETE alebo UPDATE príkazu Open SQL alebo subquery.

```
SELECT [..] FROM [..]
WHERE [..] GROUP BY [..] HAVING [..]
ORDER BY [..]
%_HINTS <selector> '<text>' <selector> '<text>' [..] .
```

Teda klauzula %_HINTS pozostáva z dvojíc databázových selektorov a textu hintu. Selektory sú kľúčové slová, a preto nie sú uvedené v apostrofoch. Možné selektory sú: ADABAS, AS400, DB2, DB6, INFORMIX, MSSQLNT, ORACLE.

Databázové rozhranie vyhodnocuje texty hintov pred ich posunutím databáze a vykonáva susbstitúcie v týchto textoch.

Pre použitie hintov je potrebné sa riadiť pokynmi z OSS note 129385 (Database hints in Open SQL) a je možné používať iba hinty uvedené v nasledovných platformovo-špecifických OSS Notes:

- 130480 Database hints in Open SQL for Oracle
- 133381 Database hints in Open SQL for MS SQL Server
- 150037 Database hints in Open SQL for DB6 (DB2 UDB)
- 152913 Database hints in Open SQL for Informix
- 162034 DB2/390: Database hints in Open SQL
- 485420 iSeries: Database hints for Open SQL/Native SQL
- 652096 Database hints in Open SQL for SAPDB / MaxDB

14 Buffering

Niektoré Open SQL dotazy nemusia pristupovať priamo do databázy, ale môžu využívať tzv. buffer. Každá inštancia aplikačného servera SAP má buffer, ktorý obsahuje nejaké dáta z databázy. V prípade, že sú tieto dáta v bufferi, k databáze pri ich čítaní nemusí byť vôbec pristupované, pretože buffer umožní priame čítanie z pamäte aplikačného servera. To má dve základné výhody:

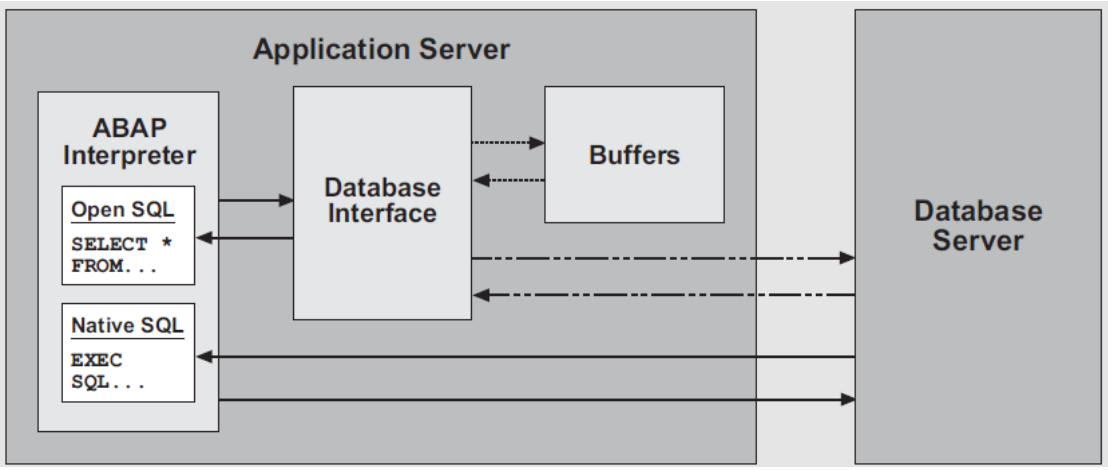
1.) Prístup do SAP bufferov je bežne 10- až 100-krát rýchlejší ako do databázy.

2.) Záťaž na databázu je zredukovaná.[11]

Viacerými meraniami sa mi podarilo potvrdiť, že rýchlosť prístupu do SAP bufferov je 10- až 100-krát rýchlejší (v mnou meraných prípadoch to bolo 15- až 60-krát), ale nemá význam tu uvádzať konkrétne merania.

Zredukovaná záťaž na databázu je dôležitá najmä v prípade veľkých systémov, pretože aplikačný server pracuje vždy s jedným databázovým serverom, ale množstvo aplikačných serverov môže byť ľubovoľné. To znamená, že presunutie časti záťaže databázového servera na aplikačnú úroveň je žiadúce.

Buffre sa bežne synchronizujú s databázou vo väčšine prípadov každých 60 sekúnd (prednastavená hodnota parametra /rsdisp/bufreftime).



11 SCHNEIDER, T. SAP Performance Optimization Guide

Existujú 3 druhy buffrovania (buffering):

1.) Buffrovanie jednotlivých záznamov – pri prístupe k záznamu je tento záznam vložený do buffera.

2.) Buffrovanie generických oblastí – pri prístupe k záznamu je celá generická oblasť zodpovedajúca tomuto záznamu vložená do buffera. Generická oblasť je definovaná počtom kľúčových polí.

3.) Kompletné buffrovanie – pri prístupe k ľubovoľnému záznamu je obsah celej tabuľky vložený do buffera.

To, ktoré tabuľky majú byť buffrované a akým spôsobom, je súčasťou prednastaveného nastavenia štandardnej dodávky systému SAP. Niekedy môže byť nutná zmena týchto nastavení z dôvodu zlepšenia výkonnosti. To má za úlohu administrátor systému. Z pohľadu programátora je dôležité konštruovať Open SQL príkazy tak, aby efektívne využívali buffrovanie tabuliek.

Príkazy obchádzajúce buffer:

· BYPASSING BUFFER dodatok k FROM klauzule

· DISTINCT dodatok k SELECT klauzule

· Agregačné výrazy v SELECT klauzule

· Joiny vo FROM klauzule

· IS NULL podmienka vo WHERE klauzule

· Subquery vo WHERE klauzule

· ORDER BY klauzula

· GROUP BY klauzula

· FOR UPDATE dodatok

· Native SQL príkazy[12]

12 http://help.sap.com/saphelp_nw70/helpdata/en/aa/4734a00f1c11d295380000e8353423/content.htm

Pre tabuľky s buffrovaním jednotlivých záznamov sú to všetky príkazy okrem SELECT SINGLE. V prípade tabuliek s generickým buffrovaním sú to všetky SQL príkazy okrem SELECT *, ak WHERE podmienka je podmienka na ekvivalenciu pre všetky polia zahrnuté v generickej oblasti.[13]

Týmto príkazom sa treba v prípade selekcie dát z buffrovaných tabuliek vyhnúť okrem prípadov, keď nám prípadná neaktuálnosť dát prekáža. To znamená, že v prípade selekcie dát z plne buffrovaných tabuliek:

1.) namiesto príkazu DISTINCT treba odstrániť duplikáty na aplikačnej úrovni pomocou príkazov SORT a DELETE ADJACENT DUPLICATES

2.) v niektorých prípadoch je lepšie výber dát rozčleniť na viac krokov namiesto použitia JOINu alebo subquery

3.) ak nie je nutné rozlišovať medzi iniciálnou a NULL hodnotou, vybrať(vymazať) riadky s iniciálnou hodnotou na aplikačnej úrovni

3.) triedenia iné ako podľa primárneho kľúča treba vykonávať na aplikačnej úrovni pomocou príkazu SORT

4.) v prípade generického buffrovania treba dávať vždy podmienku na ekvivalenciu na všetky kľúčové polia

5.) v prípade buffrovania jednotlivých záznamov treba používať SELECT SINGLE s plne špecifikovaným kľúčom

13 http://help.sap.com/saphelp_nw04/helpdata/en/f7/e4c5aba84a11d194eb00a0c929b3c3/content.htm

Čítanie buffrovaných kmeňových dát pre užíavateľsky špecifické rozšírenia

Počas transakcie alebo programu, môže byť viacero prístupov ku kmeňovým dátam pomocou plne špecifikovaného primárneho kľúča. Tieto prístupy sa vyskytujú najmä v user-exitoch alebo iných používateľských rozšíreniach. Tieto prístupy je možné nahradiť použitím čitania s buffra. V SAP štandarde na tento účel existujú viaceré funkčné moduly, ktoré si ukladajú výsledok poslednej (resp. niekoľkých posledných) databázových požiadaviek v pamäti. Potom je možné použiť túto uloženú hodnotu znova, bez potreby prístupu do databázy. Tým pádom môže byť záťaž na databázu zredukovaná.

Teda napríklad namiesto

```
SELECT SINGLE * FROM mara WHERE matnr EQ ....
```

je možné použiť funkčný modul

```
CALL FUNCTION 'MARA_SINGLE_READ'
  EXPORTING
    matnr =
  IMPORTING
    wmara =
     .
```

Pri volaní takýchto funkčných modulov systém ukladá výsledok posledného volania a výsledok je použitý znova, ak sa funkčný modul volá s identickými importnými parametrami. Pretože SAP programy tiež používajú tieto funkčné moduly, pri ich využití pri používateľských rozšíreniach sú identické databázové prístupy následne potlačené.

Zádkladné funkčné moduly pre buffrovaný prístup ku kmeňovým dátam sú tieto:

Tabuľka	Funkčný modul
MARA	MARA_SINGLE_READ
MARC	MARC_SINGLE_READ
MARD	MARD_SINGLE_READ
MAKT	MAKT_SINGLE_READ
MARM	MARM_SINGLE_READ
MBEW	MBEW_SINGLE_READ
MVKE	MVKE_SINGLE_READ
KNA1	V_KNA1_SINGLE_READ
KNB1	KNB1_SINGLE_READ
KNVV	KNVV_SINGLE_READ
LFA1	WY_LFA1_SINGLE_READ

Ak je potrebné čítať viac záznamov na jednom mieste v programe, je lepšie viaceré volania týchto funkčných modulov nahradiť jedným volaním.

Teda napríklad namiesto

```
SELECT * FROM mara INTO TABLE itab_mara
  FOR ALL ENTRIES IN itab2_mat
    WHERE matnr = itab2_mat-matnr
.
```

je možné volanie funkčného modulu

```
CALL FUNCTION 'MARA_ARRAY_READ'
  IMPORTING
    retc                 = l_number_of_errors
  TABLES
    ipre03               = itab2_mat
    mara_tab             = itab_mara
  EXCEPTIONS
    enqueue_mode_changed = 0.
```

Tieto funkčné moduly na čítanie viacerých záznamov používajú rovnakú internú pamäť ako funkčné moduly na čítanie jednotlivých záznamov. Čítanie individuálnych záznamov pomocou funkčných modulov môže teda použiť už uložené hodnoty, ktoré boli uložené funkčnými modulmi na čítanie viacerých záznamov.

Na čítanie viacerých záznamov z buffrovaných kmeňových dát existujú nasledovné funkčné moduly:

Tabuľka	Funkčný modul
MARA	MARA_ARRAY_READ
MARC	MARC_ARRAY_READ
MARD	MARD_ARRAY_READ
MAKT	MAKT_ARRAY_READ
MARM	MARM_ARRAY_READ
MBEW	MBEW_ARRAY_READ
LFA1	WY_LFA1_ARRAY_READ[14]

14 https://service.sap.com/sap/support/notes/332856

15 Využitie aktualizačných funkčných modulov

Pri transakciách, ktoré používateľ používa na vkladanie alebo modifikáciu dát v databáze, používateľ často nepotrebuje čakať, kým samotná operácia na zmenu obsahu tabuľky prebehne, ale mohol by ďalej pokračovať vo svojej práci. Preto je možné na samotnú modifikáciu obsahu tabuľky vytvoriť aktualizačný funkčný modul. Ten je potom možné zavolať v separátnom update work procese pomocou CALL FUNCTION … IN UPDATE TASK namiesto jeho vykonávania v dialógovom work procese. To znamená, že pri použití tohto dodatku nebude vykonaný okamžite, ale bude naplánovaný pre spustenie v separátnom update work procese. Tým pádom sa z pohľadu používateľa doba vykonávania skráti o celú dobu jeho vykonávania, čo môže byť výrazná doba. Aktualizačný funkčný modul nemôže obsahovať žiadne návratové parametre, pretože program z ktorého je vyvolaný nad ním nemá kontrolu. Jeho spracovanie je spustené po príkaze COMMIT WORK. V prípade, že ku commitu nepríde počas behu programu, je plánované volanie tohto funkčného modulu zrušené. Aktualizácie je možné kontrolovať pomocou transakcie SM13, teda v prípade zlyhania modifikácie obsahu databázovej tabuľky je možné s jej využitím analyzovať príčiny, prípadne sa pokúsiť aktualizáciu znovu vykonať. Nejaké porovnávanie na konkrétnych prípadoch nemá význam, pretože časová úspora z pohľadu používateľa je celá doba trvania modifikácie tabuľky.

16 Kopírovanie obsahu jednej internej tabuľky do druhej

Na prekopírovanie obsahu internej tabuľky do inej internej tabuľky existuje viacero spôsobov. Jednou možnosťou je priame priradenie:

1.)

```
lt_tab = gt_tab.
```

Ďalšími možnosťami sú priradenie riadku po riadku:

2.)

```
LOOP AT gt_tab INTO ls_tab.
  APPEND ls_tab TO lt_tab.
ENDLOOP.
```

alebo použitie príkazu APPEND LINES OF … TO:

3.)

```
APPEND LINES OF gt_tab TO lt_tab.
```

Priradenie riadku po riadku je nevyhnutné, ak je treba vykonať na niektorých poliach riadku dodatočnú operáciu, alebo tabuľka, do ktorej sa kopíruje obsah, má inú štruktúru ako zdrojová tabuľka, to znamená, že je potrebné prekopírovať hodnoty do zodpovedajúcej štruktúry.

Príkaz APPEND LINES OF … TO umožňuje určiť, ktoré riadky majú byť nakopírované pomocou špecifikovania indexu počiatočného a koncového riadka pre kopírovanie.

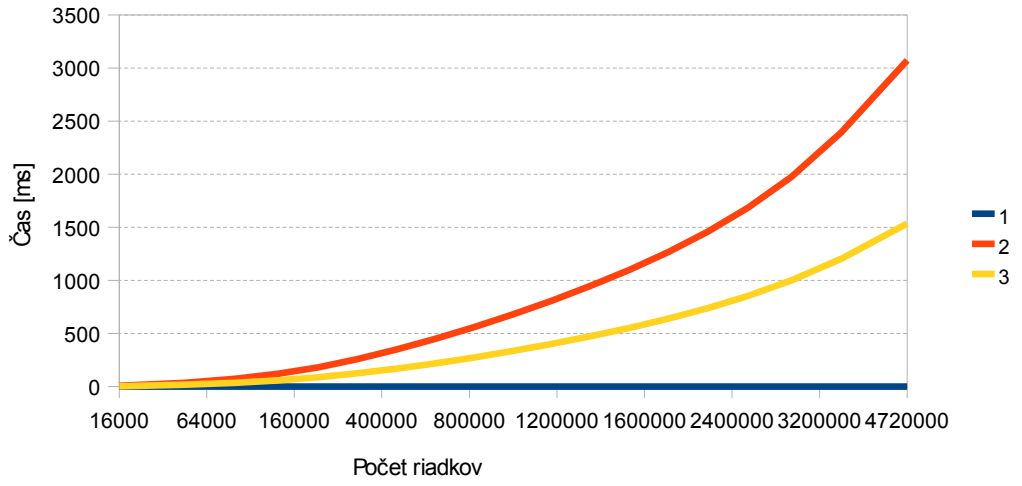
Samozrejme, najrýchlejším a najodporúčanejším spôsobom je priame priradenie, ale zaujímavé sú aj doby trvania, ktoré sa nikde neuvádzajú.

Merania som vykonával na viacerých príkladoch. Doby behu sú, samozrejme, závislé aj od nárokov na pamäť pre jeden riadok tabuľky. Čím sú

tieto nároky väčšie, tým porovnanie vychádza viac v prospech priameho priradenia, pretože pri spracovaní riadok po riadku sa musí kopírovať väčšie množstvo dát (najskôr do štruktúry a potom ako nový riadok cieľovej tabuľky). Uvádzam výsledky pre prípad, že riadok tabuľky vyžadoval 100 bytov.

Počet riadkov \ spôsob	1	2	3
16000	0,01	7,45	2,45
64000	0,01	39,05	19,24
160000	0,01	98,87	47,31
400000	0,01	246,25	119,64
800000	0,01	506,71	246,16
1200000	0,01	832,54	397,15
1600000	0,01	1028,72	527,81
2400000	0,01	1543,07	785,09
3200000	0,01	2080,65	1056
4720000	0,01	3072,73	1534,93

Tabuľka č.9 - Doby trvania behu jednotlivých spôsobov vzhľadom na počet záznamov v milisekundách.



Ako vidno z nameraných výsledkov, poradie spôsobov z hľadiska výkonnosti je podľa očakávaní. Avšak rozdiely medzi týmito spôsobmi nie sú až také výrazné. Merania som vykonával aj na prípadoch, že riadok tabuľky zaberá v pamäti iné miesto a ukázalo sa, že pri druhom a treťom spôsobe je doba trvania priamo úmerná počtu bytov, ktoré vyžaduje jeden riadok. Na základe mojich meraní by sa dala predpokladať doba trvania pre druhý spôsob približne *0,000005 x počet riadkov x počet bytov potrebných pre jeden riadok x konštanta závislá od výkonnosti systému*.

17 Záver

Tvorba efektívnych programov v jazyku ABAP je rozsiahla téma. Časť z nej sa mi podarila pokryť v mojej práci. Pri jej tvorbe som vyskúšal niektoré nové nástroje a získal nové poznatky.

Vyskúšal a popísal som možnosti využitia rôznych nástrojov pri tvorbe efektívnych programov.

Podarilo sa mi zhrnúť základné pravidlá čerpané z iných zdrojov, ako aj hlbšie sa venovať niektorým konkrétnym pravidlám.

Ukázal som možnosť využitia dynamického Open SQL pri zefektívnení programu, riziká spojené s jeho použitím. Analyzoval som možnosti využitia dynamického špecifikovania jednotlivých klauzúl pre zlepšenie výkonnosti. Venoval som sa aj analýzam časov potrebných na dodatočné parsovanie počas runtime, kde som zistil, že tieto časy sú zanedbateľné v porovnaní z dobou trvania vyhodnotenia samotného dotazu.

Tiež som ukázal možnosť využitia kurzora a indexu pri práci s internými tabuľkami pre hlavičkové a položkové dáta, rovnako ako aj časové rozdiely jednotlivých spôsobov.

Detailne som sa venoval porovnaniu využitia symbolov polí namiesto štruktúr ako pracovnej oblasti pri spracúvaní riadkov internej tabuľky. Porovnal som typizované a netypizované symboli polí. Ukázal som, že z výkonnostného hľadiska je medzi nimi priepastný rozdiel. Tiež som ukázal paradoxné zhoršenie doby behy pri znížení množstva prenášaných dát pomocou dodatku TRANSPORTING.

Porovnal som selekciu dát do internej tabuľky so selekciou do štruktúry, pričom som popísal aj teoretické rozdiely v týchto spôsoboch, ale aj priniesol konkrétne merania, ktoré ukazujú, že rozdiel v dobách trvaní pri jednotlivých spôsoboch nie je až taký priepastný, aby z tohto zaužívaného pravidla spravil nemennú dogmu a vylúčil použitie spôsobu bez selekcie dát priamo do internej tabuľky.

Venoval som sa možnostiam využitia jednotlivých typov interných tabuliek ako na teoretickej úrovni, tak aj na základe konkrétnych meraní. Ukázal som, aký značný dopad môže mať výber nesprávneho typu tabuľky na celkovú výkonnosť systému, pretože je to práve práca s internými tabuľkami, čo najviac vyťažuje aplikačné servery.

Detailne som sa venoval možnostiam kontroly existencie záznamu resp. doťahovaniu dodatočných údajov, pretože tie, keď sú vykonávané v cykloch, spôsobujú najväčší počet prístupov do databázy, čo môže do značnej miery spomaliť celý systém.

Nemalú pozornosť som venoval aj príkazu FOR ALL ENTRIES, ktorý je ABAP-špecifickým rozšírením SQL štandardu, ktorému nebola v iných zdrojoch venovaná dostatočná pozornosť – napriek tomu, že jeho nesprávne použitie môže spôsobiť obrovské výkonnostné problémy a do neúnosnej miery zaťažiť databázový server. Programátor by sa však nemal jeho používaniu vyhýbať, pretože jeho správne použitie prináša viaceré možnosti aj z hľadiska zlepšenia výkonnosti.

Tiež som ukázal, že nielen informácie technického charakteru sú užitočné pri tvorbe efektívneho kódu, ale pri vývoji pre konkrétny SAP modul sú to aj informácie o dátovom modely, ktorý obsahuje namiesto indexov pre veľa základných tabuliek indexové tabuľky, ktoré umožnia rýchlejší prístup k dátam.

Stručne som prezentoval možnosti využitia Open SQL hintov.

Venoval som sa ABAP bufferu a možnostiam jeho využitia za účelom zníženia záťaže databázy, príkazom, ktoré ho obchádzajú a spôsobom ako nahradiť niektoré prístupy, ktoré ho obchádzajú prístupami, ktoré ho využívajú.

Poukázal som na možnosť využitia SAP-om dodávaných funkčných modulov pri práci s dátami v dialógových transakciách určených pre prácu s kmeňovými dátami na zredukovanie počtu prístupov do databázy.

Spomenul som možnosť využitia aktualizačných funkčných modulov na odľahčenie dialógových procesov a tým pádom na zredukovanie doby trvania dialógovej transakcie.

Na viacerých meraniach som získal doby trvania napĺňania internej tabuľky po riadkoch pomocou príkazu APPEND, čím som ponúkol aspoň približnú predstavu o trvaní jednej zo základných operácií s internými tabuľkami.

Stále zostáva veľké množstvo možných tém a konkrétnych meraní, ktoré by mohli priniesť zaujímavé výsledky, na základe ktorých by bolo možné vyvodiť závery, avšak vzhľadom na obmedzený rozsah tejto práce nebol pre ne dostatočný priestor.

Zoznam informačných zdrojov

Görler, A., Koch, U. Enhanced ABAP Programming with Dynamic Open SQL. In: SAP Professional Journal, September/Október 2001. Dostupné v PDF verzii na internete: <http://www.sappro.com/article.cfm?session=&uid=ce077372-1655-4d58-bfac-3331553b9371>

SAP Help Support. 20.05.2008 [cit. 2009-04-25]. Dostupné na internete: <http://help.sap.com/>

SAP Service Marketplace. 13.03.2009 [cit. 2009-04-25]. Dostupné na internete: <http://service.sap.com/>

SCHNEIDER, T. *SAP Performance Optimization Guide.* Preložil Lemoine International, Inc., Salt Lake City. 4. vyd. Bonn : SAP Press, 2009. 624 s. ISBN 159229202X.

SCHWARZ, W. *Performance Problems in ABAP Programs: How to Find Them.* In: SAP Professional Journal, Máj/Jún 2003. Dostupné v PDF verzii na internete: <http://www.sappoint.com/PHPWebUI/Documents/Performance%20Problems%20in%20ABAP%20Programs%20How%20to%20Find%20Them.pdf>