TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Analýza kódu ABAP v systému v SAP a automatická optimalizace

Diplomová práce

Studijní program: N2612 Elektrotechnika a informatika
Studijní obor: Informační technologie

Autor práce: **Bc. Jan Procházka**
Vedoucí práce: Ing. Lenka Kosková Třísková, Ph.D.
Ústav nových technologií a aplikované informatiky

Liberec 2021

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Zadání diplomové práce

Analýza kódu ABAP v systému v SAP a automatická optimalizace

Jméno a příjmení:	**Bc. Jan Procházka**
Osobní číslo:	M18000151
Studijní program:	N2612 Elektrotechnika a informatika
Studijní obor:	Informační technologie
Zadávající katedra:	Ústav nových technologií a aplikované informatiky
Akademický rok:	**2020/2021**

Zásady pro vypracování:

1. Navrhněte a implementujte metodu měření rychlosti běhu programu v prostředí SAP.
2. Navrhněte a realizujte analyzátor kódu v SAP, který nalezne předem určené neoptimální dotazy a struktury.
3. Navrhněte popis případných doplňujících pravidel pro strukturu kódu, jež budou uplatněny při optimalizaci.
4. Vytvořte optimalizační program, který nalezené části kódu nahradí dle předem navržených pravidel.
5. Hotové řešení testujte na předem zvolené množině kódu. Testy dokumentujte.
6. Změřte rychlost původního a optimalizovaného kódu a výsledky srovnejte.

Rozsah grafických prací: dle potřeby dokumentace
Rozsah pracovní zprávy: 40 – 50 stran
Forma zpracování práce: tištěná/elektronická
Jazyk práce: Čeština

Seznam odborné literatury:

[1] Mogensen, T. A.: Introduction to Compiler Design, Springer-Verlag London Limited 2011, e-ISBN 978-0-85729-829-4
[2] Watson, D: A Practical Approach to Compiler Construction,Springer International Publishing AG 2017, ISBN 978-3-319-52789-5
[3] Grune, D. et all: Modern Compiler Design, Springer Science+Business Media New York 2012, ISBN 978-1-4614-4699- 6
[4] Parr T.: The Definitive ANTLR Reference, The Pragmatic Bookshelf, 2007, ISBN: 0-9787392-5-6.
[5] Bandari K.: Complete ABAP, SAP Press, 2017, ISBN 978-1-4932-1273-6.

Vedoucí práce: Ing. Lenka Kosková Třísková, Ph.D.
Ústav nových technologií a aplikované informatiky

Datum zadání práce: 19. října 2020
Předpokládaný termín odevzdání: 17. května 2021

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

Ing. Josef Novák, Ph.D.
vedoucí ústavu

V Liberci dne 19. října 2020

Prohlášení

Prohlašuji, že svou diplomovou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Jsem si vědom toho, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS/STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má diplomová práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

17. května 2021 Bc. Jan Procházka

Poděkování

Rád bych poděkoval své vedoucí Ing. Lence Kostkové Třískové za vedení a rady při mé diplomovém práci. Dále bych rád poděkoval Ing. Lukáši Sýkorovi ze společnosti T-MC66, s.r.o. za poskytnutí testovacího klienta od společnosti SAP. A v neposlední řadě bych rád poděkoval své rodině, bez které bych studium nezvládl.

Abstrakt

Cílem práce bylo zjistit možnosti snížení využité paměti a doby běhu programu v jazyce ABAP v systému SAP, realizovat nástroj pro automatickou optimalizaci a následně změřit reálnou změnu těchto prostředků na optimalizovaném zdrojovém kódu. Pro měření byl implementován vlastní měřící program, který porovnával systémový čas a využití paměti před spuštěním programu a po ukončení jeho běhu. Největší úspora využité paměti byla prokázána při čtení dat z databázové tabulky, a to až o 90 procent. Největší úspora doby běhu programu byla prokázána při čtení dat z interní tabulky. Použitím vhodného typu interní tabulky lze zkrátit dobu běhu programu až o 4 řády.

Klíčová Slova:

SAP, ABAP, spotřeba paměti, doba běhu, gramatika ABAP, optimalizace.

Abstract

The aim of the work was to determine the possibilities of reducing the used memory and program run time written in ABAP language for system SAP, to implement a tool for automatic optimization and then to measure the real change of these resources on the optimized source code. For the measurement, our own measuring program was implemented, which compared the system time and memory usage before starting the program and after its end. The greatest memory savings were demonstrated when reading data from a database table, by up to 90 percent. The greatest saving in program runtime was demonstrated when reading data from an internal table. By using a suitable type of internal table, the program run time can be reduced by up to 4 orders of magnitude.

Keywords:

SAP, ABAP, memory usage, runtime, ABAP grammar, optimalization

Obsah

Úvod 11

TEORETICKÁ ČÁST 12

1 SAP 12

1.1 Systémy pro plánování podnikových zdrojů 12

1.2 Současná situace 12

1.3 Společnost SAP 13

1.4 Produkty 14

2 ABAP 15

2.1 Struktura programu 16

2.2 Práce s databází 17

2.3 Práce s interní tabulkou 17

2.4 Procházení interní tabulky 18

3 Automatická optimalizace 19

3.1 Výběr dat z databáze 19

3.2 Zpracování dat 20

3.3 Uchování dat 21

4 Metody měření a dostupné nástroje v SAPu 22

PRAKTICKÁ ČÁST 23

5 Přeměření jednotlivých struktur 23

5.1 Měření a výsledky 24

6 Diskuze k měření optimalizace 28

7 Optimalizátor 30

7.1 Strom programu 30
7.2 Převod kódu na strom programu 31
7.3 Gramatika jazyka ABAP a testovaní 33
7.3.1 Rozsah nové gramatiky 34
7.4 Gramatika optimalizátoru 34
7.5 Optimalizace stromu programu 36
7.5.1 Optimalizace výběru dat z databáze 37
7.5.2 Optimalizace zpracování dat při běhu programu 37
7.5.3 Optimalizace prefixu proměnných 38
7.6 Převod stromu programu na kód 39
7.7 Testovaní funkčnosti 39
7.8 Rozšiřování optimalizátoru 40
8 Metoda měření optimalizátoru 41
9 Výsledky měření 42
9.1 Výsledky měření před optimalizací 42
9.2 Výsledky měření po optimalizaci 43
10 Diskuze k výsledkům optimalizace za použití optimalizátoru 45
Závěr 46
Příloha A – obsah repositáře 50

Seznam grafů

Graf 1 - Využití ERP systémů v roce 2019 [2] 13
Graf 2- srovnání rychlostí operací nad různými typy interních tabulek 20
Graf 3 - porovnání rozdílu měření před optimalizací a po optimalizaci 44

Seznam obrázků

Obrázek 1 - Diagram databázové struktury 23
Obrázek 2 - Zjednodušené schéma procesu optimalizace 30
Obrázek 3 - Syntaktický strom pro referenční kód 32
Obrázek 4 - Strom programu pro referenční kód 32
Obrázek 5 - Diagram optimalizace příkazu SELECT 37
Obrázek 6 - diagram optimalizace prefixu proměnné 38

Seznam tabulek

Tabulka 1 - Porovnání velikosti testovacích sad 24
Tabulka 2 - Výsledky měření "SELECT *..." 25
Tabulka 3 - Výsledek měření "SELECT col, col ..." 25
Tabulka 4 - Výsledky operace JOIN 25
Tabulka 5 - Výsledky měření pro SELECT zanořený v cyklu 25
Tabulka 6 - Výsledky měření podmíněného výběru dat v cyklu 26
Tabulka 7 - Procházení interní tabulky pomocí "INTO" 26
Tabulka 8 - Procházení interní tabulky pomocí "ASSIGNING" 26
Tabulka 9 - Doba běhu operace READ z interní tabulky [µs] 26
Tabulka 10 - Doba běhu operace DELETE z interní tabulky [µs] 26
Tabulka 11 - Doba běhu operace INSERT do interní tabulky [µs] 27
Tabulka 12 - Úloha 1 před optimalizací 42
Tabulka 13 - Úloha 2 před optimalizací 42
Tabulka 14 - Úloha 3 před optimalizací 42

Tabulka 15 - Úloha 4 před optimalizací 42
Tabulka 16 - Úloha 5 před optimalizací 43
Tabulka 17 - Úloha 1 po optimalizaci 43
Tabulka 18 - Úloha 2 po optimalizaci 43
Tabulka 20 - Úloha 4 po optimalizaci 44
Tabulka 21 - Úloha 5 po optimalizaci 44
Tabulka 22 - Úloha 6 po optimalizaci 44

Seznam zdrojových kódů

Zdrojový kód 1 - Program pro vlastní měření prostředků 22
Zdrojový kód 2 - Referenční kód pro porovnání stromů 32
Zdrojový kód 3 - Porovnání původní a nové gramatiky 33
Zdrojový kód 4 - Sekce řízení optimalizátoru před a po optimalizaci 35
Zdrojový kód 5 - Gramatika optimalizátoru 35
Zdrojový kód 7 - ukázka převodu uzlu stromu programu na zdrojový kód 39
Zdrojový kód 6 - ukázka převodu stromu programu zpět na zdrojový kód 39
Zdrojový kód 8 - Ukázka testovací úlohy pro pravidlo Assigment 40

Úvod

SAP je jedna ze softwarových nadnárodních společností, které vytvářejí systémy pro plánování podnikových zdrojů (tzv. ERP systémy). Tyto systémy pokrývají veškeré firemní procesy nezbytné pro chod podniku, např. plánování logistiky, evidenci účetnictví, skladového hospodářství a mnoho dalších. ERP systémy umožňují podnikům centralizovat veškeré informace do jednoho systému. Díky tomu může být část lidských zdrojů využita na jiné činnosti. Dále není potřeba školit zaměstnance v používání a údržbě více systémů a tím se snižuje riziko zanesení chybných dat do systémů. Z tohoto důvodu se ERP systémy využívají zejména ve velkých firmách, kde je potřeba řídit značné množství lidí a procesů, tedy v místech, kde by mohlo dojít k desinformaci či ztrátě dat.

Tyto systémy díky dlouhodobému vývoji obsahují prověřené procesy, které vedou k vysoké efektivitě práce. Na druhou stranu použitím těchto systémů firma může ztratit své unikátní procesy a snižuje se tím její konkurenceschopnost. Proto SAP, který vyvíjí ERP sytém SAP Bussiness Suite, umožňuje upravit systém dle vlastních požadavků. Jelikož je psán vlastním programovacím jazykem, musí tyto úpravy provádět specializovaná vývojářská firma. Avšak ani tyto firmy neprovádí vývoj vždy v dostatečné kvalitě. To následně může vést ke zpomalení programu, zbytečným ztrátám paměti a případě až k pádu samotné aplikace. Existuje několik řešení návrhu programu, jak se těmto ztrátám vyvarovat, ale ne každý vývojář je používá. Z tohoto důvodu značná část firem provádí ještě dodatečnou kontrolu kvality kódu a jeho čitelnosti.

Jelikož kontrola kvality kódu je časově náročný proces, rozhodl jsem se vytvořit nástroj, který provádí tuto kontrolu automaticky a následně opravuje nalezené problémy. Dále je využit i pro jiné kontroly nad zdrojovým kódem, např. kontrolu a optimalizaci čitelnosti kódu. Nejprve jsem pomocí měření vyhodnotil, která místa jsou vhodná pro optimalizaci a poté vytvořil nástroj, který vychází z procesu překladů zdrojového kódu. Zdrojový kód je nejprve převeden na syntaktický strom. K tomu byla použita gramatika jazyka ABAP a využit nástroj ANTLR. Syntaktický strom je následně převeden na strom programu. Jedná se o jiný druh stromu, který je vhodný pro tuto problematiku. Ve stromu programu jsou provedeny všechny kontroly a realizovány případné optimalizace. Upravený strom programu je nakonec převeden zpět na zdrojový kód, čímž je proces optimalizace ukončen. Tato optimalizace vedla ke zlepšení doby běhu a využití paměti programem, což bylo ověřeno měřením výsledného zdrojového kódu.

TEORETICKÁ ČÁST

1 SAP

1.1 Systémy pro plánování podnikových zdrojů

Systém pro plánování podnikových zdrojů je systém, kterým firma řídí a integruje většinu podnikových oblastí či procesů. Název pochází z anglického pojmu *Enterprise Resource Planning*, místo kterého se běžně používá zkratka ERP. Podnikové oblasti jsou například plánování, zásoby, nákup, prodej, marketing, finance, personalistika, výroba atd.

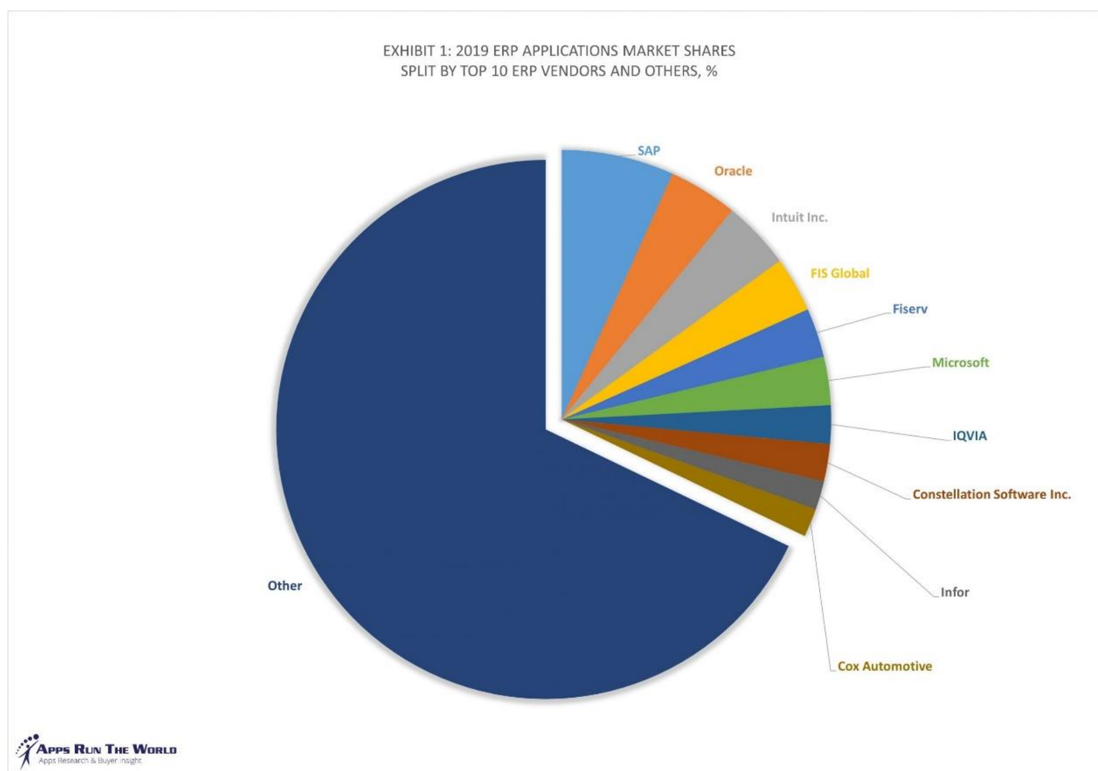
Při výběru systému ERP jsou dvě možnosti. První je zvolit kompletní řešení a druhou je nechat si naimplementovat systém podle vlastních požadavků. Výhodou první možnosti je, že tyto systémy jsou velmi kvalitní a mnoha zákazníky prověřené. Umožňují tak zefektivnit a standardizovat firemní procesy. Na druhou stranu – spolu s přejímáním standardních procesů firma ztrácí i své unikátní procesy a tím ztrácí i konkurenceschopnost.

Druhou možností je nechat si upravit či vytvořit systém přímo podle firemních požadavků. Tato volba je ovšem časově i finančně náročnější, avšak vlastní systém umožňuje ponechat si své firemní procesy [1].

1.2 Současná situace

Jednou z nejúspěšnějších firem, která se zabývá vývojem ERP systémů je společnost SAP. Dalšími přední výrobci těchto systémů na světovém trhu jsou zejména Oracle a Microsoft. Situace z roku 2019 je vidět na Grafu 1 [2].

Na českém trhu je přes 100 firem, které vyvíjejí ERP systémy. Ty jsou však cíleny především na český trh, a proto nepatří mezi světovou špičku. Mezi nejvýznamnější ERP systémy patří HELIOS a K2 [3].



Graf 1 - Využití ERP systémů v roce 2019 [2]

1.3 Společnost SAP

SAP je německá firma, která byla založena v roce 1972 bývalými zaměstnanci z IBM. Jejich cílem bylo vyvinout podnikový informační systém, který by obsahoval co nejvíce firemních funkcí. Z počátku vyvíjeli aplikace pro správu financí a skladů. V 90. letech produkty společnosti SAP používalo více než 70 % velkých německých firem. Největší úspěch zaznamenala tato firma nasazením produktu R/3 v roce 1992. V roce 2015 přišel SAP s novou verzí – S/4 HANA.

Mezi jejich současné zákazníky (mimo ČR) patří např. Walmart, Apple, Shell, General Motors, Ford Motor atd. Mezi v česku působící firmy patří například Škoda auto, a.s.; Johnson Controls, k.s.; Magna Automotive, s.r.o.; ČEZ, a.s.; RWE, a.s.; T-Mobile, a.s.; Česká pošta, s. p.; Agrofert Holding, a.s.; Preciosa, a.s.; Okay, a.s.; AVAST Software, s.r.o. nebo Karlovarské minerální vody, a.s.

SAP nabízí vedle svých produktů také konzultační a implementační služby. Pražská centrála také zajišťuje podporu a konzultační služby pro východní Evropu a severní Afriku [2, 4].

1.4 Produkty

Nejznámějším produktem firmy SAP je ERP systém pojmenovaný Bussines Suite. Jedná se již o dříve zmíněný produkt SAP R/3, kterému končí podpora v roce 2025, a jeho novější verzi S/4 HANA (tyto produkty jsou často nepřesně označovány pouze jako SAP). Dnes již SAP nabízí upravené verze těchto systémů, aby byly použitelné pro střední a malé firmy. Nutno zmínit, že přestože podpora starého systému končí za čtyři roky, stále se ve velké míře využívá, vytváří se v něm nové projekty a firmy vyčkávají s přechodem na novější verzi na poslední chvíli.

Dále firma SAP nabízí přes 300 dalších produktů a služeb. Některé z těchto položek jsou propojeny s Bussines Suite, jiné jsou na ní naopak naprosto nezávislé. Nejvýznamnější službou je SAP Cloud platform, nástroj pro šíření a sdílení firemních aplikací. Dále lze za významný považovat i analytický nástroj Bussines Intelligence. Ten slouží pro analýzu chování zákazníků, odhadů metrik či analyzování a následné odstranění odhadů při nastavování firemních procesů [4].

2 ABAP

ABAP je programovací jazyk, který vznikl pro programování aplikací od společnosti SAP. První verze vznikala v 70. letech 20. století z jazyka assembler. V roce 2000 byl rozšířen o objektové programování. Dnes jsou tímto jazykem programovány veškeré aplikace SAP Bussiness Suite, kromě jádra, které je psáno v jazyce C. Od roku 2003 je možno programovat zmíněné aplikace také v jazyce JAVA [5].

Objekty jazyka ABAP a názvosloví

Každý vývojář musí bezpodmínečně znát pro rozšiřování systému jeho základní objekty. Těmito základními objekty jsou: paket, databázová tabulka, datový prvek, doména, struktura, program, třída, funkce, funkční modul, funkční skupina, transakce atd. V následujícím výčtu jsou pouze objekty relevantní pro rozsah této práce.

Paket

Každému objektu v SAPu přiřazujeme paket. V každém paketu se vytváří skupina objektů, se kterou lze hromadně pracovat – přesouvat mezi servery. Označení pro lokální (test-vývoj) server je $TMP. Jedná se o obdobu „Package“ z jazyka Java.

Databázová tabulka

Je základem celého systému SAP. Jsou v nich uchována veškerá data celého systému od nastavení až po zákaznická data. Množství tabulek SAPu se pohybuje v řádech desetitisíců až statisíců. Tyto tabulky rozdělujeme podle typu na tři druhy – transparentní, clusterové a tabulky poolu.

Transparentní tabulky

Do těchto tabulek se ukládají zejména data aplikací.

Tabulky poolu a clusterové tabulky

V těchto tabulkách je uloženo zejména nastavení, pořadí procesů a dokumentace.

Doména a datový prvek

Doména popisuje datový prvek. Obsahuje o něm veškeré informace – datový typ, velikost (počet míst), rozsah hodnot. Datový prvek je konkrétní záznam z tabulky, který je popsán klíčem a doménou.

Struktura

Je skupina interních polí v programu, které jsou vzájemně propojeny. Data jsou uchovávána pouze po dobu chodu programu.

Dynpro

Je podobrazovka GUI, které je zpracovávána programem. Slouží pro zobrazení dat a interakci s uživatelem.

Programy

Programy rozdělujeme na dva typy. První jsou spustitelné programy – mohou obsahovat všechny procesní bloky s výjimkou funkčních modulů. Veškeré spustitelné programy musí začínat příkazem REPORT a názvem programu. Druhou skupinou jsou programy typu Include. Tyto programy slouží pouze jako moduly pro jiné programy.

Transakce

Posledním důležitým objektem je Transakce. Transakce spouští program v SAPu. SAP má definováno již velké množství transakcí pro standardní operace. Transakci můžeme přiřadit i uživatelskému programu, ta má však omezení v názvu [5–7].

2.1 Struktura programu

Programy ABAP obsahují dvě části – deklarační a funkční část. V deklarační části se deklarují veškerá data, tabulky, pole atd. Dále se zde deklarují i výběrové obrazovky se všemi parametry. Jako první se deklarují tabulky pomocí klíčového slova TABLES. Následně se poté deklarují ostatní proměnné a struktury pomocí klíčového slova DATA. Proměnné jsou pojmenovány podle specifických pravidel. První znak musí být písmeno a může být dlouhý maximálně 30 znaků.

Bloky zpracování (funkční část) obsahují jednotlivé algoritmy zpracování dat. Funkční část se obvykle skládá z více bloků zpracování. Standardně tyto bloky začínají a končí klíčovými slovy.

Pořadí těchto událostí je pevně dané. Program postupně vyhledává jednotlivé bloky v pořadí [5–7]:

1. LOAD-OF-PROGRAM – provede se okamžitě po spuštění programu, nemá žádná omezení v obsahu.
2. INITIALIZATION – v této události se provádí deklarace dat pro výběrové obrazovky nebo tuto obrazovku upravují.
3. AT-SELECTION-SCREEN – slouží zejména pro složitější kontroly vstupu. Například po potvrzení výběrové obrazovky provede kontrolu, zda uživatel zadal platné hodnoty a případně vyvolá chybné hlášení a donutí uživatele zadat hodnoty znovu.
4. START-OF-SELECTION – pokud jsou všechna data „v pořádku“, systém se dostane k události START-OF-SELECTION, kde už by měl být výpis reportu, který se uživateli na základě zadaných dat zobrazí.

2.2 Práce s databází

Pro práci s databází se využívá příkaz SELECT pro čtení, INSERT pro vkládání nebo UPDATE pro aktualizování tabulky. ABAP má možnost využít příkaz MODIFY, který aktualizuje daný záznam, případně ho založí. Nutné podotknout že pomocí těchto příkazů nepracujeme přímo s databází. Těmito příkazy pouze voláme příslušné operace na pozadí systému. Z toho plyne, že systém je nezávislý na typu databáze. Při změně dodavatele databáze se pouze upraví skripty na pozadí. Veškeré programy jsou také nezávislé na typu databáze a není potřeba upravovat veškeré programy po změně databáze [5–8].

2.3 Práce s interní tabulkou

Aby bylo možno pracovat v programu s databází je nutno ji nahrát nejdříve do interní tabulky. V současné době rozlišuje ABAP tři typy interních tabulek – STANDARD, SORTED a HASHED. Výhodou typu STANDARD je, že lze přidávat nové záznamy na konec pomocí klíčového slova APPEND. Zároveň nemusí mít každý záznam v tomto typu tabulek jedinečný klíč. Na druhou

stranu nedosahuje velkých rychlostí přístupu, protože při hledání jednoho konkrétního záznamu je nutno kontrolovat záznam po záznamu. Typ tabulky HASHED vyžaduje unikátní klíč. Je pak schopen velmi rychle přistupovat ke konkrétním záznamům. Tabulky typu SORTED využívají binární vyhledávání, protože záznamy jsou seřazeny podle hodnot klíče. Tabulky typu HASH používají pro rychlé vyhledávání konkrétního záznamu hashovací funkci [5–8].

2.4 Procházení interní tabulky

Při systematickém procházení interní tabulky lze v ABAPu využít dvou druhů cyklů – LOOP INTO a LOOP ASSIGNING. Pomocí LOOP INTO DATA je docíleno při každém průchodu cyklu vytvoření obrazu záznamu do lokální proměnné. Provedené změny do tohoto lokálního obrazu je nutné distribuovat zpět pomocí klíčového slova MODIFY. Druhou možností je použít LOOP ASSIGNING FIELD-SYMBOL. Tímto příkazem je docíleno při každém průchodu cyklu vytvoření odkazu na konkrétní záznam a při práci s FIELD-SYMBOLem je v tomto případě pracováno přímo se záznamem [5–8].

3 Automatická optimalizace

Hlavním důvodem, proč tuto problematiku řešit, je nedostatek prostředků přidělených uživateli. Přestože server disponuje dostatečným výpočetním výkonem a značným množstvím paměti, je nutné si uvědomit, že v jednu chvíli pracuje na serveru obrovské množství uživatelů. Aby se server vyhnul pádu, přiděluje jednotlivým uživatelům pouze určitou část paměti a výkonu. V případě, že uživateli dojde přidělená paměť nebo dosáhne časového limitu pro jeden program, systém okamžitě ukončí činnost daného uživatele. Je tedy nutné docílit toho, aby se veškeré výpočty stihly provést v rámci těchto limitů.

Druhým důvodem je činnost uživatelského rozhraní. Bohužel v tomto směru není systém SAP zcela optimální. Neumí provádět operace na pozadí. To znamená, že v případě výpočtu dojde k zamrznutí uživatelského rozhraní.

Příčinou těchto problémů jsou špatné programátorské návyky. Většinou se jedná o špatné čtení dat, které se obvykle vyskytuje ve dvou formách: čtení z databáze a čtení z interní tabulky.

Jednou z možností, jak se zmíněným problémům vyvarovat, je navyšovat limity pro jednotlivé uživatele. Limity jsou nastavovány systémovým správcem na každém serveru. Tuto možnost není ovšem možné aplikovat do nekonečna. Proto je nutné optimalizovat činnost programů, aby nedocházelo ke zbytečným ztrátám paměti nebo se neprodlužovala doba běhu.

Základem optimalizace je vybrat vhodné místo. Bylo provedeno 16 různých měření, které pokrývaly oblasti: výběr dat z databáze, jejich uchování po dobu běhu programu, a také jejich zpracování. Následující odstavce shrnují výsledky těchto měření a doporučení k optimalizaci. [9–11]

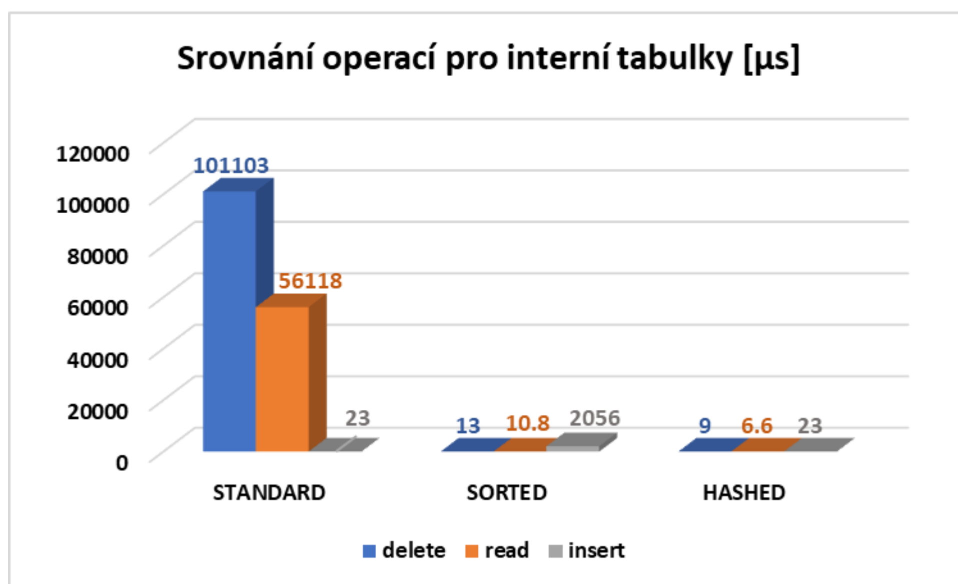
3.1 Výběr dat z databáze

Systém SAP uchovává téměř všechna data v databázi – od nastavení systému, přes aplikace až po uživatelská data. Proto je při psaní aplikací nutné dbát na optimalizaci SQL dotazů. Nejzákladnější chybou je příkaz „SELECT“. Značná část programátorů kvůli pohodlnosti vybírá všechny sloupce pomocí „SELECT *“. Zabraňuje to nutnosti přepisovat SELECT v průběhu implementace po zjištění, že chybí sloupec z databáze, který je nutno použít. Tyto příkazy nejsou ovšem efektivní. Program pak spotřebovává více prostředků, než je ve skutečnosti potřeba.

Správným postupem je z databáze vybírat pouze potřebné sloupce. Přesto se příkaz „SELECT *“ vyskytuje v 80 % případů. Zde je tedy první místo optimalizace. [9]

3.2 Zpracování dat

Při zpracování dat je možno procházet množinu dat pomocí příkazu LOOP. Data se ukládají do lokální proměnné nebo pomocí odkazu. V tomto případě nebyl zaznamenán velký rozdíl v době běhu. Rozdíl je patrný pouze ve využití paměti, kde se porovnává velikost jednoho řádku z vybraných dat a velikost odkazu. Jelikož je pak vytvořena pouze jedna instance řádku, která je posléze plněna daty, je tento rozdíl z celkového hlediska zanedbatelný. Toto místo tedy není vhodné pro optimalizaci. [9]



Graf 2- srovnání rychlostí operací nad různými typy interních tabulek (zdroj vlastní)

3.3 Uchování dat

Jak bylo zmíněno výše v systému SAP existují 3 druhy interních tabulek – standardní, tříděná a hashovaná. Každá má své výhody a nevýhody. Při uložení stejných dat pomocí každého tohoto typu není patrný velký rozdíl v použití paměti. Na druhou stranu je zde zásadní rozdíl v době běhu programu. Porovnání využití paměti a doby běhu pro tyto druhy interních tabulek je v grafu 2. Patrné je že využití standardní interní tabulky není vhodné pro velkou množinu dat, kvůli rychlosti čtení. Z tohoto ohledu je nejrychlejší metoda hashování klíče. Tento druh interní tabulky ale vyžaduje unikátní klíč tabulky, proto je nejvýhodnější při optimalizaci použít typ tabulky SORTED. [9]

4 Metody měření a dostupné nástroje v SAPu

SAP již umožňuje měřit spotřebované prostředky. Existuje několik programů, které jsou k dispozici. Základní programy jsou volány pomocí transakcí – ST05 a SAT. Každý z těchto programů je pak zaměřen na jinou oblast. Transakce ST05 obsahuje poměrně komplexní nástroje na ladění dotazů SQL a jejich vývoje. Transakce SAT měří spotřebu paměti a dobu běhu programu. Komplikací u obou transakcí je, že je nutno nejdříve spustit měření. Systém pak zaznamenává veškeré aktivity uživatele. Ty se teprve po vypnutí zapíšou do hromadného záznamu, kde se dají procházet a filtrovat. [12]

Další možností je měřit přímo požadované sekvence kódu. Existuje třída cl_abap_memory_utilities, která obsahuje metody pro získání aktuálního systémového času a aktuální spotřeby paměti. Výhodou této možnosti je měřit libovolnou část kódu jako celek. Naopak v případě, že je nutno znát dílčí hodnoty pro každý příkaz, nemá toto použití dostatečnou přesnost. Další nevýhodou je, že při měření paměti se zaznamená pouze počáteční a konečný stav. Nedojde tak k zaznamenání výkyvů využití paměti mezi počátečním a koncovým měřením. Přesto je při vhodném použití možnost rychle změřit jednotlivé části programu manuálně výhodná. Ukázka využití vlastního měření je zobrazena ve Zdrojovém kódu 1.

```
1. data datatable type table of sbook.
2. data l_count type int4.
3.
4. CALL METHOD cl_abap_memory_utilities=>get_total_used_size
5.     IMPORTING
6.       size = data(sizebefore).
7. GET RUN TIME FIELD data(tb).
8.
9.        ***********
10.       * DO SOMETHING
11.       ***********
12.
13.       GET RUN TIME FIELD data(ta).
14.       CALL METHOD cl_abap_memory_utilities=>get_total_used_size
15.           IMPORTING
16.             size = data(sizeafter).
17.       DATA(tm) = CONV decfloat34( ta - tb ).
18.       data(mem) = sizeafter - sizebefore.
19.       write: / | time is: { tm } micS |.
20.       write: / | mem is: { mem } kB |.
```

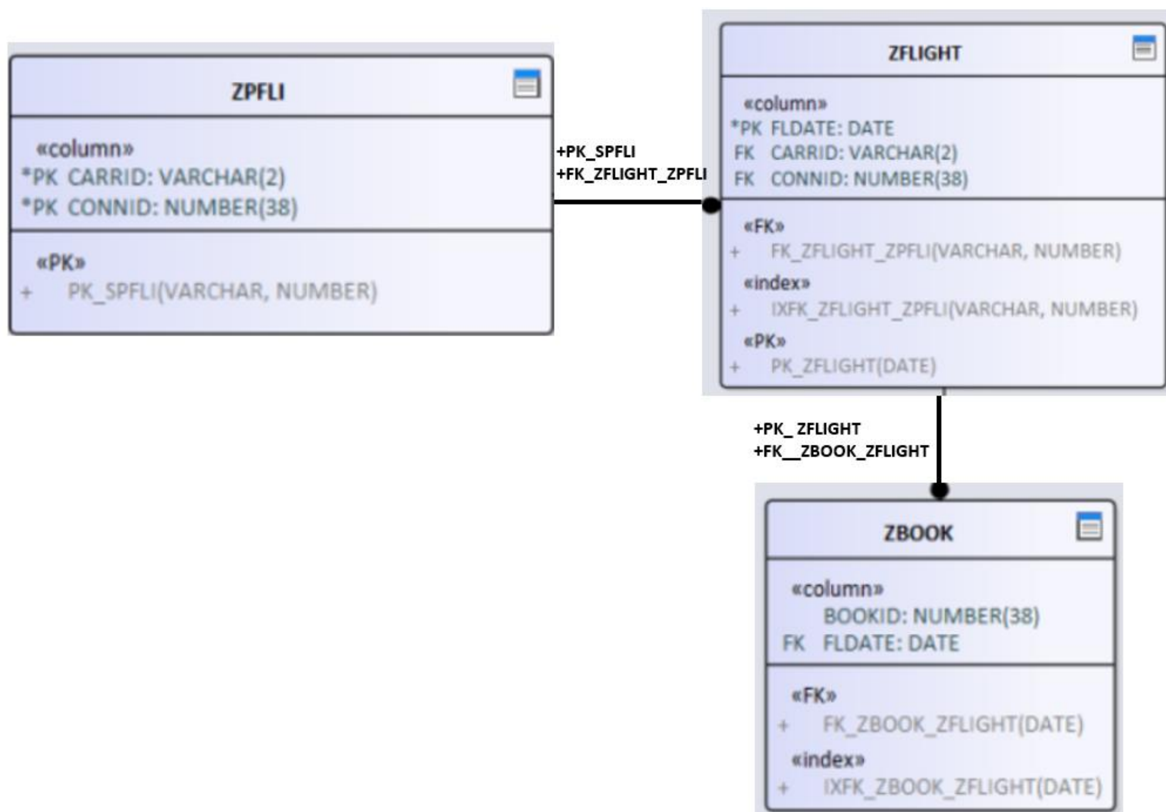
Zdrojový kód 1 - Program pro vlastní měření prostředků

PRAKTICKÁ ČÁST

5 Přeměření jednotlivých struktur

V rámci diplomového projektu na téma *Měření výkonosti vybraných struktur jazyka ABAP* [9] jsem prováděl měření vybraných struktur. Avšak ta nejsou dostatečně přesná. Byla prováděna pouze na vzorku dat o rozsahu 100 000 řádků. Proto je potřeba ověřit výsledky nad větší množinou dat.

Pro měření byl zvolen samostatný server s dostatečnými zdroji – 16GB RAM, 350 GB HDD a procesor *QEMU virtual CPU* o frekvenci 2 GHz. Na serveru v době měření nebyl přítomen žádný jiný uživatel a neprobíhaly žádné plánované operace na pozadí. Tím bylo dosaženo optimálních podmínek pro opakovatelnost měření a jejich porovnání. Pro tvorbu testovacích dat byl použit generátor zmíněný v kapitole 4. Tento generátor vytváří obsah tří tabulek: SBOOK, SFLIGHT a SPFLI. Zjednodušené schéma těchto tabulek je k dispozici na obrázku 1.



Obrázek 1 - Diagram databázové struktury

Pro testování byla použita tabulka SFLIGHT. Jedná se o největší tabulku. Po použití generátoru obsahuje více než 1.3 milionu záznamů. Oproti tomu staré měření bylo prováděno nad databází s pouhými 100 000 záznamy. Porovnání velikostí všech tabulek je v tabulce 1. Jak je vidět obsah prvních dvou tabulek, tj. SPFLI a SPFLIGHT se zdvojnásobil a obsah největší tabulky se více než zdesetinásobil.

Tabulka 1 - Porovnání velikosti testovacích sad

Počet záznamů v tabulce	První sada	Druhá sada
SPFLI	26	46
SPFLIGHT	26	46
SBOOK	101000	1326000

Samotné měření bylo prováděno pomocí vlastního měřícího programu. Důvodem bylo zabránit ovlivnění měření inicializací proměnných. Ukázka takového programu je ve Zdrojovém kódu 1.

5.1 Měření a výsledky

Samotné měření bylo prováděno na stejných úlohách, jaké byly použity v *Měření výkonosti vybraných struktur jazyka ABAP* [9]. Byla tedy použita stejná měření:

1. SELECT – byla změřena spotřeba prostředků v případě, kdy se vybírají všechny sloupce a v případě, kdy se vybírají pouze potřebné sloupce
2. JOIN – byla měřena spotřeba prostředků při výběru 3 sloupců ze 2 tabulek
3. SELECT IN LOOP – byla měřena spotřeba prostředků scénáře, kdy se nejprve vyberou 2 sloupce z první tabulky a poté se v cyklu vybírá jeden sloupce z druhé tabulky
4. LOOP INTO DATA – byla měřena spotřeba prostředků při procházení interní tabulky pomocí klausule INTO
5. LOOP ASSIGNING FIELD-SYMBOL – byla měřena spotřeba prostředků při procházení interní tabulky pomocí klausule ASSIGNING
6. STANDARD TABLE – byla změřena spotřeba prostředků při standardních operacích INSERT, DELETE a UPDATE

7. SORTED TABLE – byla změřena spotřeba prostředků při standardních operacích INSERT, DELETE a UPDATE
8. HASHED TABLE – byla změřena spotřeba prostředků při standardních operacích INSERT, DELETE a UPDATE

Každé z těchto měření mělo svůj účel. Měření 1. – 3. byla zaměřena na výběr dat z databáze, měření 6. – 8. měřila rychlost přístupu k datům v průběhu programu a měření 4. a 5. měřilo rychlost práce s daty.

Zbytek kapitoly shrnuje výsledky jednotlivých měření. Tyto výsledku jsou dále diskutovány v následující kapitole.

V tabulkách 2 a 3 jsou výsledky měření 1.

*Tabulka 2 - Výsledky měření "SELECT *..."*

ZSELECT_COLS	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
doba běhu [µs]	787681	851236	803250	737632	749476	**785855**	*45358.3*
využití paměti [kB]	18382168	18382168	18382168	18382168	18382168	**18382168**	*0*

Tabulka 3 - Výsledek měření "SELECT col, col ..."

ZSELECT_STAR	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
doba běhu [µs]	7255155	7228943	7167988	7227528	6977801	**7171483**	*112881*
využití paměti [kB]	357450368	357450368	357450368	357450368	357450368	**357450368**	*0*

V tabulkách 4, 5 a 6 jsou výsledky měření 2 a 3.

Tabulka 4 - Výsledky operace JOIN

ZJOIN	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
doba běhu [µs]	979287	1070232	879719	1000612	952136	**976397**	*69525*
využití paměti [kB]	17081272	17081272	17081272	17081272	17081272	**17081272**	*0*

Tabulka 5 - Výsledky měření pro SELECT zanořený v cyklu

ZJOIN	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
doba běhu [µs]	77781753	89854209	80360994	79153501	81624659	**81755023.2**	*4746365*
využití paměti [kB]	17054152	17054152	17054152	17054152	17054152	**17054152**	*0*

Tabulka 6 - Výsledky měření podmíněného výběru dat v cyklu

ZJOINSELECT	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
doba běhu	1115664	1100578	1108495	1101913	1151312	**1115592.4**	*20849.6*
využití paměti	17054152	17054152	17054152	17054152	17054152	**17054152**	*0*

V tabulce 7 jsou zobrazeny výsledky pro měření 4 a v tabulce 8 jsou výsledky měření 5.

Tabulka 7 - Procházení interní tabulky pomocí "INTO"

ZLOOP	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
LOOP into data							
doba běhu [µs]	138254	139433	138375	141294	140054	**139482**	*1260*
využití paměti [kB]	0	0	0	0	0	**0**	*0*

Tabulka 8 - Procházení interní tabulky pomocí "ASSIGNING"

ZLOOP	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
LOOP asigning data							
doba běhu [µs]	102374	105350	102969	105701	105261	**104331**	*1538*
využití paměti [kB]	136	136	136	136	136	**136**	*0*

V tabulce 9 jsou výsledky pro operaci READ, v tabulce 10 pro operaci DELETE a v tabulce 11 jsou výsledky pro operaci INSERT.

Tabulka 9 - Doba běhu operace READ z interní tabulky [µs]

ZTABLES	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
standard table	54702	57910	54656	59201	54120	**56118**	*2283*
Hashed table	6	6	7	7	7	**6.6**	*0.5*
Sorted table	11	11	11	11	10	**10.8**	*0.4*

Tabulka 10 - Doba běhu operace DELETE z interní tabulky [µs]

ZTABLESDELETE	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
Standard table	99812	102412	101835	100946	100510	**101103**	*1036*
Hashed table	*9*	*9*	*9*	*8*	*9*	**8.8**	*0.4*
Sorted table	13	12	13	13	13	**12.8**	*0.4*

Tabulka 11 - Doba běhu operace INSERT do interní tabulky [μs]

ZTABLESINSERT	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
Standard table	22	25	24	23	22	**23.2**	*1.3*
Hashed table	23	*23*	*24*	*24*	*21*	**23**	*1.2*
Sorted table	2274	1772	2533	2135	1617	**2066**	*372*

Četnost jednotlivých struktur byla převzata z *Měření výkonosti vybraných struktur jazyka ABAP* [1].

- Celkem programů: 12326
    - SELECT: 2315
        - SELECT *: 1804
        - SELECT konkrétních sloupců: 511
    - SELECT dat z více tabulek: 126
        - JOIN: 84
        - SELECT v LOOPu: 42
    - Interních tabulek: 3686
        - STANDARD: 3591
        - SORTED: 61
        - HASHED: 34

6 Diskuze k měření optimalizace

Při měření 1 byla porovnána spotřeba paměti pro příkaz „SELECT *“ a „SELECT sloupec, sloupec, …“. Při porovnání výsledků z tabulky 2 a 3 je patrné, že došlo k výraznému zrychlení při výběru konkrétních sloupců oproti výběru všech sloupců. V tomto konkrétním případu došlo až k 10násobnému zrychlení doby běhu programu a v oblasti využití paměti došlo ke 35násobnému ušetření prostředků. Důvodem je, že tabulky v systému SAP obvykle obsahují desítky sloupců. Naopak při implementaci se často využívají pouze sloupce v řádu jednotek. Definovat zde přesnou míru zlepšení není ale možné, protože toto číslo je závislé na počtu vybíraných sloupců a na jejich datovém typu. Přesto lze jednoznačně říci, že se jedná o dobré místo pro optimalizaci.

Měření 2 a 3 (viz tabulka 4 a 5) potvrdilo obecný předpoklad, že spojování dat z více tabulek je rychlejší na úrovni databáze než za použití následného zpracování. Nicméně praxe ukazuje, že se dodatečný výběr dat ve smyčce neustále používá. Toto místo jsem určil jako vhodné pro optimalizaci. Měřením jsem určil hodnotu rovnosti příkazu JOIN a podmíněného výběru dodatečných dat, která nastává přibližně okolo 100. cyklu (*IF sy-tabix mod 100 = 0*). Zde došlo k výraznému zhoršení oproti první testovací sadě, kdy rovnost nastávala pro podmínku *(IF not sy-tabix mod 3 = 0)*.

Měřením 4 jsem zjistil množství spotřebovaných prostředků při procházení interní tabulky pomocí LOOP INTO. Dochází zde k lokálnímu vytváření obrazu procházeného řádku. Jak je z Tabulky 7 vidět využití paměti je 0 kB. To je způsobeno tím, že se instance lokálních dat po konci cyklu smaže, tzn. před cyklem a po cyklu je využití paměti stále stejné.

V měření 5, viz Tabulka 8, se porovnávala rychlost a využití paměti za použití cyklu „LOOP ASSIGNING“, které pracuje přímo s daty. V tomto případě se vytvoří field-symbol (odkaz – pointer) na konkrétní řádek v interní tabulce.

Při porovnání měření 4 a 5 je zřejmé, že rychlejší je měření 5. Vytváření field-symbolu je tedy nepatrně rychlejší, než vytváření obrazu dat. Rozdíl mezi těmito metodami je ale vzhledem k celkové době běhu zanedbatelný a nejedná se o dobré místo k optimalizaci.

Nejzajímavější výsledky jsou z měření 6. – 8., kde porovnávám způsoby uložení dat po dobu běhu programu a jejich zpracování. Konkrétně měření 6 zaznamenává prostředky pro operaci READ, měření 7 pro operaci DELETE a měření 8 pro operaci INSERT.

Tabulka 9 zaznamenává výsledky pro měření 6, tj. operaci READ. Na první pohled je jasné, že nejčastěji používaný typ interní tabulky STANDARD má ovšem výrazně pomalejší dobu čtení, a to až o 4 řády. Naopak typy HASHED a SORTED jsou si velmi blízko.

V tabulce 10 jsou výsledky 7. měření – operace DELETE. Zde je patrná podoba s Tabulkou 9. To je dáno tím, že operace READ a DELETE jsou postaveny na stejném základu vyhledávání správného řádku. Oproti operaci READ je operace DELETE nad typem interní tabulky STANDARD ještě výrazně pomalejší. Typy HASHED a SORTED dosahují opět dobrých výsledků.

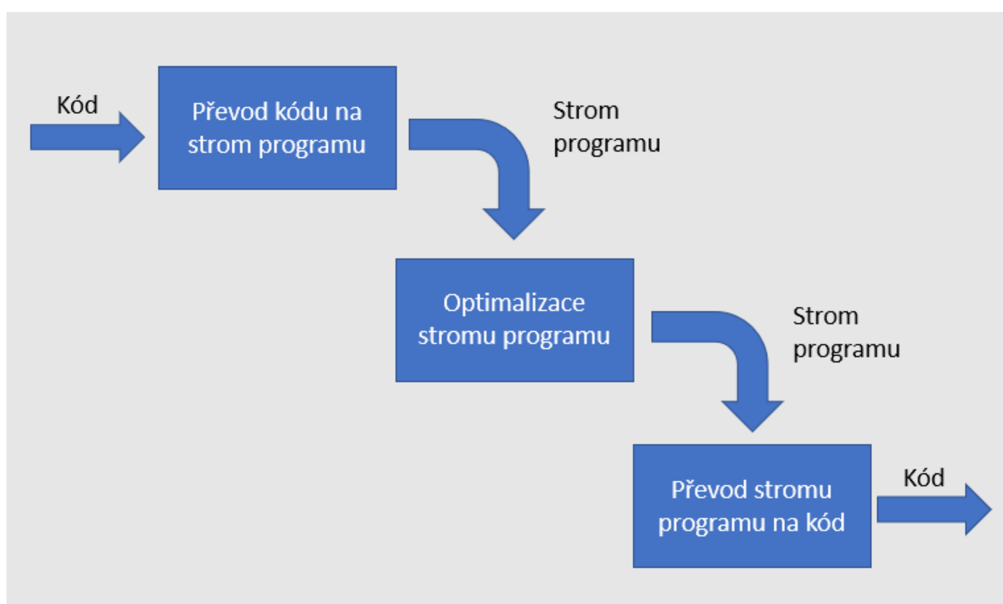
Poslední operací je INSERT. Výsledky pro tuto operaci jsou v tabulce 11. V tomto případě dosahuje typ interní tabulky STANDARD naopak velmi dobrých výsledků. Je to dáno tím, že v typu STANDARD se data vkládají na konec. To samé platí i pro typ HASHED. Naopak typ SORTED dosahuje v tomto případě nejhorších výsledků. Důvodem je, že se nový řádek musí zatřídit na správné místo.

Měřením bylo prokázáno, že typy STANDARD a SORTED mají naprosto shodnou velikost. Typ HASHED má nepatrně větší nároky na paměť, což je způsobeno ukládáním i hash klíče. Tento rozdíl je ovšem zanedbatelný, a naopak v oblasti rychlosti předčí tento typ zbylé dva. Bohužel má ale tento typ omezení ve využití, a proto je v rámci automatické optimalizace lepší využívat typ SORTED.

7 Optimalizátor

Stěžejní částí mé diplomové práce je vytvořit nástroj pro automatickou optimalizaci zdrojových kódu v jazyce ABAP pro systém SAP. Tento program jsem pojmenoval optimalizátor a jeho části vychází z teorie překladačů[13–15].

Činnost optimalizátoru, který jsem vytvořil, lze rozdělit do několika bodů: analýza zdrojového kódu, vytvoření stromové struktury reprezentující daný kód, optimalizaci stromu, konverze ze stromu na zdrojový kód. Tento algoritmus je patrný na následujícím diagramu (Obrázek 2).



Obrázek 2 - Zjednodušené schéma procesu optimalizace

7.1 Strom programu

Strom programu je reprezentace kódu programu pomocí stromové struktury. Výhodou tohoto uspořádání je jednoduché uchování zanoření dílčích částí programu – nadřazený uzel odpovídá nadřazenému bloku kódu a naopak. Problém s přehazováním bloků kódu je tak převeden na základní operace pro manipulaci se stromy.

Základním typem uzlu tohoto stromu je struktura InfixNode, která reprezentuje libovolnou část kódu. Jedná se o interface, který definuje vnitřní strukturu každého bloku a další uzly, např. LoopNode, ConditionNode atd., jej pak implementují. Tím je zaručeno, že každý takovýto uzel

bude mít implementované metody OptimizeRead a ToString. To je důležité proto, že metoda OptimizeRead obsahuje optimalizaci pro konkrétní typ uzlu při optimalizaci READ a metoda ToString obsahuje jeho převod zpět na kód.

7.2 Převod kódu na strom programu

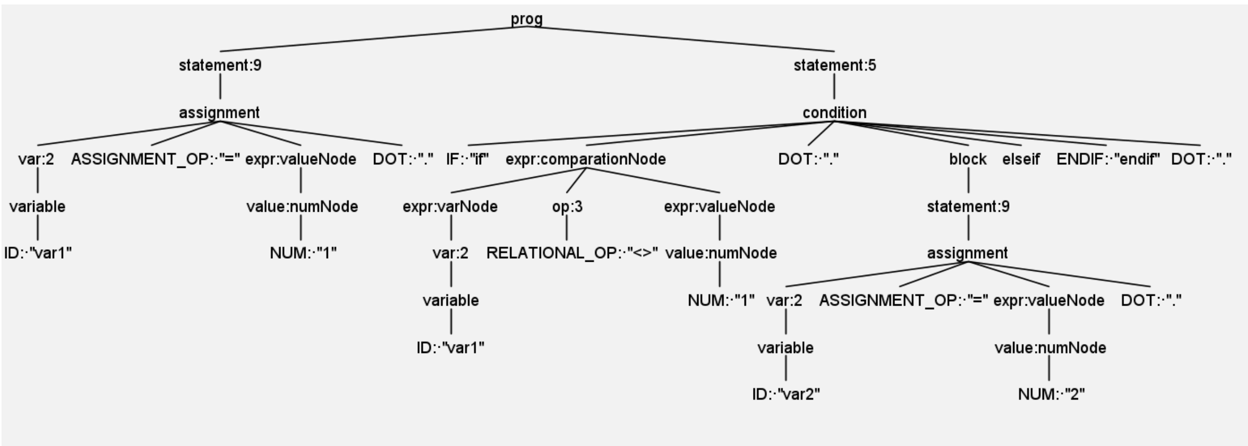
Pro vytvoření stromu programu je použita gramatika jazyka ABAP. Výstupem této gramatiky je syntaktický strom. Při následném procházení toho stromu, je postupně vytvářen strom programu. Tento strom již obsahuje pouze uzly typu InfixNode. Důvodem, proč je nutno vytvořit strom programu, je, že syntaktický strom nemusí odpovídat podobě kódu. V syntaktickém stromu zanoření totiž odpovídají zanoření pravidel v gramatice. Proto není nutné při každém takovémto zanoření vytvářet nový uzel do stromu programu. Strom programu pak odpovídá zanoření kódu v programu, což je smysluplnější. Na Obrázku 3 je znázorněný syntaktický strom pro Zdrojový kód 3 a na Obrázku 4 je znázorněn strom programu pro stejný zdrojový kód. Při porovnání těchto dvou stromů je patrné, že strom programu má čistší strukturu. Nejsou zde nadbytečná zanoření, protože značné množství zanoření je převedeno na atribut nadřazeného uzlu.

Další činností, která se provádí v průběhu vytváření stromu programu je registrace vybraných typů uzlů do optimalizátoru. V tomto konkrétním případě se jedná o typy uzlu *Select*, *Variable, Read* a *Data*. Dělá se to proto, aby nebylo nutné ihned na začátku optimalizačního procesu, prohledávat celý strom za účelem zjištění počátečního uzlu dílčí optimalizace. Tj. místo, aby se prohledával celý strom a pokaždé, když se najde uzel typu *Select*, se pro něj provedla optimalizace, tak program použije pouze připravený seznam těchto uzlů.

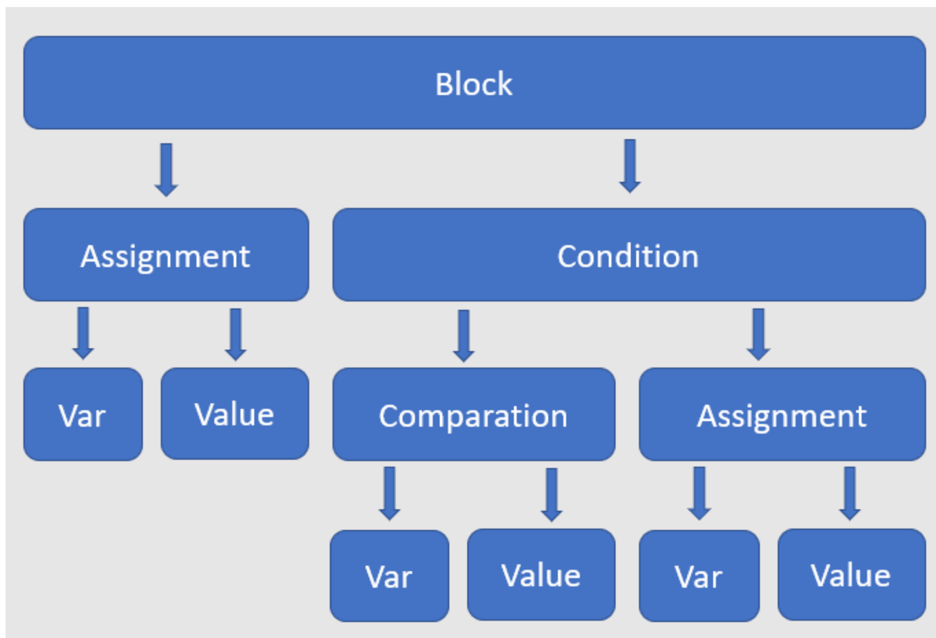
Další výhodou třídění uzlů do skupin podle typu je rychlost optimalizace. V současné době jsou realizovány pouze optimalizace, které nevyžadují procházení stromu. Tzn. pro jejich optimalizaci není potřeba rozlišovat úroveň zanoření. Strom programu je ovšem nepostradatelný pro zpětný převod na kód a pro případné další optimalizace.

```
1. var1 = 1.
2. if var1 <> 1.
3. var2 = 2.
4. endif.
```

Zdrojový kód 2 - Referenční kód pro porovnání stromů



Obrázek 3 - Syntaktický strom pro referenční kód



Obrázek 4 - Strom programu pro referenční kód

7.3 Gramatika jazyka ABAP a testovaní

Jak jsem již zmínil, gramatika jazyka ABAP je stěžejní částí pro tvorbu stromu programu. V tomto bodě jsem chtěl využít již hotovou gramatiku od Ing. Jana Vacka. Bohužel se postupně ukázalo, že tato gramatika není pro tyto účely vhodná. Gramatika byla napsána přímo pro řešení převodu mezi starou a novou verzí jazyka ABAP. Také nebyla použita jiná než základní syntaxe, což značně komplikuje procházení syntaktického stromu, který je jejím výstupem po aplikaci na kód. Rozhodl jsem se tedy starou gramatiku přepsat do nové syntaxe. Nakonec jsem byl nucen napsat vlastní gramatiku, která využívá novou syntaxi a je uzpůsobena mé problematice. Názorné rozdíly mezi starou a novou gramatikou jsou patrné ve zdrojovém kódu 3.

Jelikož napsat celou gramatiku jazyka ABAP nebylo cílem této práce, realizoval jsem pouze dílčí část gramatiky. Rozsah této gramatiky byl zvolen tak, aby pokrýval základní struktury programovacího jazyka a bylo tedy možno provádět vybrané optimalizace. Nejedná se tak o kompletní gramatiku jazyka ABAP.

```
Původní gramatika

    1.  SORT : 'sort' | 'SORT';
    2.  SORTABLE : 'sortable' | 'SORTABLE';
    3.  SORTED : 'sorted' | 'SORTED';

    4.  read_assign :
    5.  READ TABLE user_defined
    6.  ASSIGNING FIELDSYMBOL LPAREN user_defined RPAREN DOT;

Nová gramatika
    1.  SORT : [sS][oO][rR][tT] ;
    2.  SORTABLE : [sS][oO][rR][tT][aA][bB][lL][eE] ;
    3.  SORTED : [sS][oO][rR][tT][eE][dD] ;

    4.  readtable:
    5.  READ TABLE table=var  ( rtabk=rtable_key | idx=index )?
    6.  ( ptype=INTO | ptype=ASSIGNING | ptype=REFERENCE  INTO |
        ptype=TRANSPORTING NO FIELDS)
    7.  result=var
    8.  topt=transport_options? cast=casting?
    9.  DOT
    10. ;
```

Zdrojový kód 3 - Porovnání původní a nové gramatiky

Posledním důležitým parametrem gramatiky je schopnost validace kódu. Jelikož překládaný kód pochází ze systému SAP, je možno prohlásit, že optimalizátor bude provádět pouze optimalizaci *aktivních* programů. Proto jsem nemusel zohledňovat kontrolu validity kódu při vytváření nové gramatiky.

7.3.1 Rozsah nové gramatiky

Nová gramatika je psána jako ta původní v nástroji ANTLR [16, 17]. Tento nástroj totiž umožňuje testovat gramatiku v reálném čase a není potřeba před testem něco generovat. Pro testování gramatiky byly vytvořeny testovací scénáře. Každému scénáři pak odpovídá jeden testovací soubor dostupný v repositáři.

Nová gramatika umožňuje v současné době rozšiřovat následující příkazy:

- REPORT
- LOOP
- IF
- SELECT
- READ TABLE
- CLEAR

Dále pak programové konstrukty:

- Přiřazení
- Komentář
- Deklaraci dat – standardní a inline
- Prázdný řádek – v tomto případě znamená prázdný řádek pouze výskyt samotného ukončovacího znaku. Bílé znaky jsou gramatikou ignorovány.

7.4 Gramatika optimalizátoru

Důležitou vlastností optimalizátoru je možnost řídit jeho chod. Bez této možnosti by optimalizátor nebylo možné využívat v tak velkém měřítku. Představte si třeba situaci, kdy je vyžadováno optimalizovat pouze SQL dotazy. V tom případě je již nežádoucí dělat jiné optimalizace.

Vzniká tedy potřeba řídit optimalizační proces. První možností je parametrizovat celý optimalizátor. Při spuštění se pomocí nepovinných atributů nastaví jeho chování. Jak ale docílit

toho, aby se nemusel celý optimalizátor nastavovat před každým použitím znovu? I tento proces jsem se tedy rozhodl automatizovat. Optimalizační proces se nastavuje již v rámci zdrojového kódu pomocí komentáře.

Hlavní výhodou tohoto postupu je to, že řídící příkazy jsou viditelné již ve zdrojovém kódu a zároveň neovlivňují chod programu. Po optimalizaci se v tomto bloku také nachází informace

```
1. //sekce ve zdrojovém kódu před optimalizací
2. *optimalization
3. *optimize prefix prf_
4. *endoptimalization
5.
6. //sekce ve zdrojovém kódu po optimalizaci
7. *optimized 14.2.2021
8. *optimalization
9. *optimize prefix prf_
10.      *endoptimalization
```

Zdrojový kód 4 - Sekce řízení optimalizátoru před a po optimalizaci

```
1.        grammar optimalizator;
2.
3.        STAR: [*];
4.        OPTIMIZE : [oO][pP][tT][iI][mM][iI][zZ][eE];
5.        SELECT: [sS][eE][lL][eE][cC][tT];
6.        PREFIX: [pP][rR][eE][fF][iI][xX];
7.        READ: [rR][eE][Aa][dD][tT][aA][bB][lL][eE];
8.        OPTIMALIZATION:
          [oO][pP][tT][iI][mM][aA][lL][iI][zZ][aA][tT][iI][oO][nN] ;
9.        ENDOPTIMALIZATION:
          [eE][nN][dD][oO][pP][tT][iI][mM][aA][lL][iI][zZ][aA][tT][iI][oO][nN]
          ;
10.       ID: [a-zA-Z_0-9]*;
11.       WS  :   [' '\t\r\n] -> channel(HIDDEN);
12.
13.       prog: optimal body=block endoptimal;
14.       optimal : STAR OPTIMALIZATION;
15.       endoptimal: STAR ENDOPTIMALIZATION;
16.       block: ( star | select | prefix | read)*;
17.       prefix: STAR OPTIMIZE PREFIX prf=ID;
18.       star : STAR;
19.       read: STAR OPTIMIZE READ;
20.       select: STAR OPTIMIZE SELECT #selectOpt ;
```

Zdrojový kód 5 - Gramatika optimalizátoru

o výsledku optimalizace. To je důležité zejména kvůli uchování informací o nastavení optimalizace. V případě nespokojenosti s optimalizací lze pak jednoduše určit co bylo optimalizováno a je možno to v relativně krátké době vrátit zpět. Ve Zdrojovém kódu 4 v horní části je ukázka řídících direktiv pro optimalizátor a ve spodní části je ukázka řídícího bloku po optimalizaci.

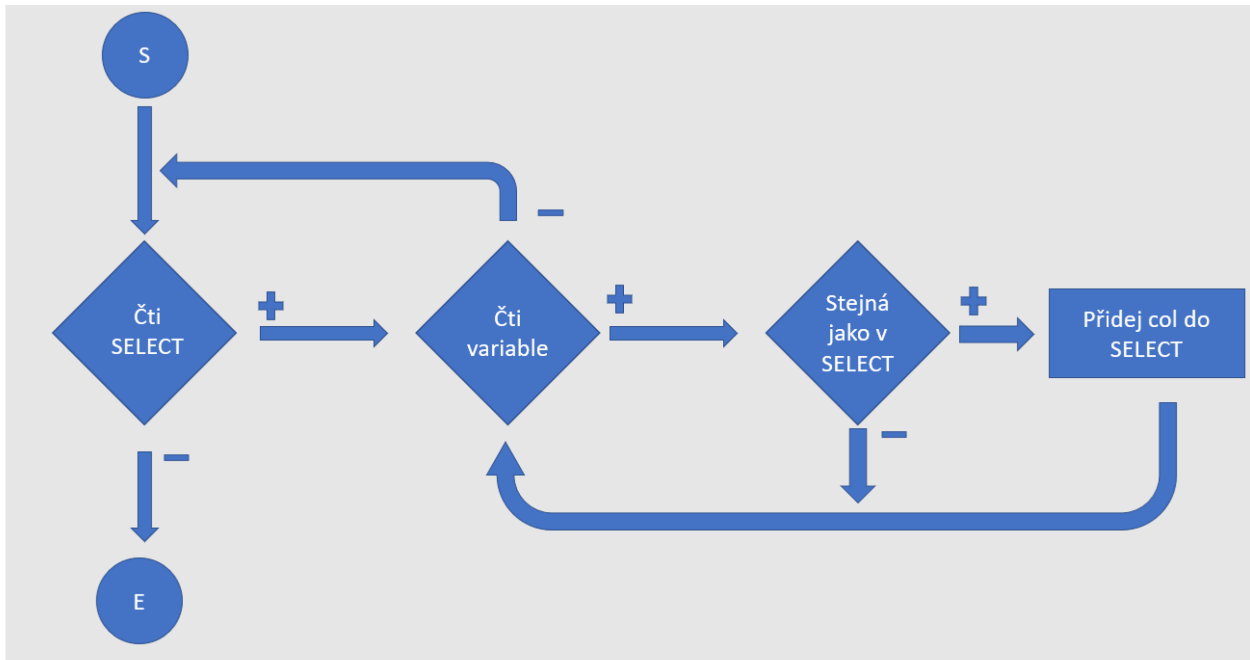
Vznikla tedy další gramatika – gramatika optimalizátoru. Jedná se o jednoduchou, relativně krátkou gramatiku, která slouží k nastavení optimalizátoru. Tuto gramatiku zaznamenává Zdrojový kód 5. Celý proces je tak automatizován. V rámci diplomové práce tato gramatika rozpoznává pouze tři řídící příkazy – *optimize select*, *optimize prefix* a *optimize read*.

7.5 Optimalizace stromu programu

Jak je zmíněno výše, v rámci této diplomové práce jsou implementovány pouze tři optimalizace. Optimalizace výběru dat z databáze, optimalizace čtení dat z interní tabulky a optimalizace zaměřená na opravu prefixu proměnných. První dvě úlohy reprezentují optimalizaci výkonu. Třetí reprezentuje optimalizaci čitelnosti kódu.

7.5.1 Optimalizace výběru dat z databáze

Tato optimalizace používá připravený seznam objektů Select, aby nebylo potřeba procházet celý strom programu a nad každým selectem provádí optimalizaci. Tato optimalizace je zaměřena především na využití paměti - *„select * ...“* nahrazuje *„select sloupec1 sloupec2 ...“*. Při této optimalizaci optimalizátor zjistí, jaké sloupce jsou v programu doopravdy využity a následně provede optimalizaci výběru dat z databáze. Celý algoritmus je znázorněn na Obrázku 5.



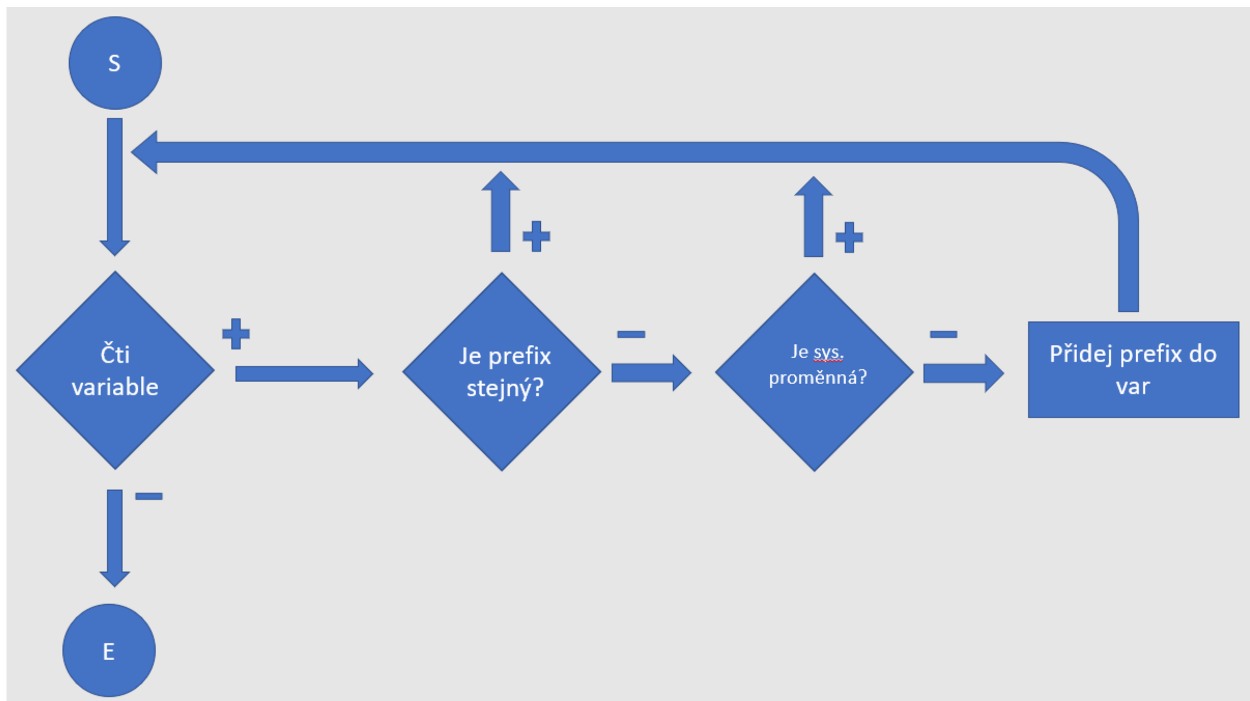
Obrázek 5 - Diagram optimalizace příkazu SELECT

7.5.2 Optimalizace zpracování dat při běhu programu

Výchozím objektem pro tuto optimalizaci je uzel READ. Místo prohledávání celého stromu opět využívá předpřipravený seznam těchto objektů. Samotný algoritmus je následující. Pro každý uzel READ určí odkud se čtení provádí. Toto místo je opět uzel typu InfixNode, konkrétně jeho implementace VarNode. Následně se pro tuto proměnnou vyhledá její deklarace (typ uzlu Datanode). V tomto místě se provádí samotná optimalizace. Předchozím měřením (Kapitola 5) bylo rozhodnuto, že nevhodnějším typem interní tabulky je typ SORTED. Dojde tedy k úpravě vnitřních hodnot uzlu DataNode, aby typ tabulky odpovídal typu SORTED.

7.5.3 Optimalizace prefixu proměnných

Tato optimalizace je zaměřena na udržitelnost kvality kódu zejména v rozsáhlých projektech, kde se očekává větší počet různých programátorů. Optimalizace prochází předem připravený seznam proměnných a kontroluje, jestli je dodržená jmenná konvence daného projektu. Z praxe jsem vypozoroval, že se tak děje převážně pomocí prefixu, a proto i v tomto konkrétním případě se používá prefix. Opětovně je celý algoritmus této optimalizace znázorněn na Obrázku 6.



Obrázek 6 - diagram optimalizace prefixu proměnné

7.6 Převod stromu programu na kód

Po ukončení optimalizace je nutno převést strom programu zpět na text. Díky stromové struktuře je tento problém velmi jednoduchý. Díky tomu, že každý typ uzlu stromu má implementovanou metodu ToString stačí převést vrchol na text, dceřiné uzly jsou pak automaticky převedeny také. Ukázka jednoduchosti převodu na text je ve Zdrojovém kódu 7 a ukázka metody ToString najdete ve Zdrojovém kódu 6. V tomto případě se jedná o uzel LOOP.

```
1. @Override
2. public String toString() {
3.     StringBuilder sb = new StringBuilder();
4.     sb.append("LOOP" + from.toString());
5.     if(to.fsymb){
6.         sb.append("ASSIGNING");
7.     }else{
8.         sb.append("TO");
9.     }
10.          sb.append( to.toString() + ".\n");
11.          for (InfixExpressionNode statement : statements){
12.              sb.append(statement.toString());
13.          }
14.          sb.append("ENDLOOP.\n");
15.          return sb.toString();
16.      }
```

Zdrojový kód 6 - ukázka převodu uzlu stromu programu na zdrojový kód

```
1. BufferedWriter writer = new BufferedWriter(new
   FileWriter(file_path.replace(".txt","out.txt")));
2. writer.write(ast.toString());
3. writer.close();
```

Zdrojový kód 7 - ukázka převodu stromu programu zpět na zdrojový kód

7.7 Testovaní funkčnosti

Testování optimalizátoru bylo provedeno na sadě testovacích příkladů. Tato sada se skládá, jak z jednoduchých příkladů, tak i z komplexnějších. Nejedná se ale o validní zdrojový kód jazyka ABAP. Jsou to pouze bloky kódu v náhodném pořadí, aby se určila správná funkčnost optimalizátoru. Celou tuto sadu můžete najít v repositáři. Úlohy testují elementární funkčnost

optimalizátoru, tj. jestli je konkrétní optimalizace správně provedena. Ukázka testovací úlohy je ve Zdrojovém kódu 8.

```
1.  pom = value1.
2.  pom = 234.
3.  pom = '234'.
4.  pom = 3 + 5.
5.  pom = pom2 - pom3.
6.  pom = 234 - pom2.
7.  <pom> = value1.
8.  <pom> = 234.
9.  <pom> = '234'.
10. <pom>-col1 = 3 + 5.
11. <pom>-col1 = pom2 - pom3.
12. <pom>-col1 = 234 - pom2.
13. <pom> = <pom2>-col1.
```

Zdrojový kód 8 - Ukázka testovací úlohy pro pravidlo Assigment

7.8 Rozšiřování optimalizátoru

Jak jsem již zmínil výše, realizace celé gramatiky jazyka ABAP je nad rámec diplomové práce, a tak pro použití v praxi je potřeba tento projekt dále rozšířit. Po celou dobu vývoje, jsem kladl důraz na čistý koncept a rozšiřitelnost. Rozšiřování je tedy možné realizovat ve dvou směrech: rozšiřování gramatiky jazyka ABAP a rozšiřování optimalizace.

- Pro rozšiřování gramatiky je dobré použít následující postup:
  1. Úprava gramatiky ABAP
  2. Úprava generování stromu programu + vytvoření nových typů uzlů

  Úprava gramatiky s sebou nese nutnost vygenerovat zní v nástroji ANTLR nové soubory, které je potřeba upravit, aby umožňovaly pracovat s typem uzlu *InfixNode*. Vytvoření nových typů uzlu zahrnuje také jejich převod na kód.

- Pro rozšíření optimalizace je potřeba implementovat samotnou optimalizaci. Pro tu lze využít funkci *optimize*, která existuje v každém typu uzlu. Dále je potřeba rozšířit gramatiku optimalizátoru, aby šla nová optimalizace použít.

8 Metoda měření optimalizátoru

Za účelem testování jsem vytvořil 6 testovacích scénářů. Jedná se o komplexní úlohy měřící úspory prostředků na validních příkladech. Tyto úlohy dále testují, zda optimalizátor optimalizuje pouze požadované části a zda neoptimalizuje již optimalizované části.

- Úloha 1 – Jedná se o úlohu, která optimalizuje všechny možnosti, tj. *select*, *prefix* a *read*. Tato úloha obsahuje systémové proměnné, které mají systémový prefix „sy-“. Tyto proměnné by neměly být optimalizovány. Pro ostatní proměnné by měl být použit prefix „prf_“.
- Úloha 2 – V této úloze probíhá kontrola, zda se neprovádí optimalizace prefixu pro proměnné, které již prefix „l_“ obsahují. Dále je zde použita také optimalizace *select*.
- Úloha 3 – testuje optimalizace *select*. Obsahuje také *read* v cyklu, který se neoptimalizuje.
- Úloha 4 – provádí opět všechny optimalizace. Obsahuje systémové proměnné. Jako prefix byl použit „*pref*“. Na rozdíl od předešlé úlohy obsahuje více příkazů *select*.
- Úloha 5 – Tato úloha kontroluje, zda se provádí správně optimalizace *read*. Testovací příklad obsahuje všechny dostupné struktury, která současná gramatika dovoluje použít.
- Úloha 6 – V této úloze se provádí optimalizace *select*. Úloha obsahuje více příkazů *select* a kontroluje, zda se optimalizace navzájem neovlivňují.

Stejně jako měření optimalizace v kapitole 5 byla i tato měření provedena za konstantních podmínek popsaných dříve. Zmíněná měření byla provedena dvakrát – jednou před optimalizací a podruhé po optimalizaci.

9 Výsledky měření

9.1 Výsledky měření před optimalizací

Tabulka 12 obsahuje měření úlohy 1 před optimalizací.

Tabulka 12 - Úloha 1 před optimalizací

ZTEST1	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
doba běhu [µs]	49931923	47562900	51111375	47853362	51131160	**49114890**	*1698393*
využití paměti [kB]	357460752	357460752	357460752	357460752	357460752	**357460752**	*0*

Tabulka 13 obsahuje výsledky měření úlohy 2 před optimalizací.

Tabulka 13 - Úloha 2 před optimalizací

ZTEST2	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
doba běhu [µs]	5852130	5735087	6251245	6407526	6215301	**6061497**	*319436*
využití paměti [kB]	273093360	273093360	273093360	273093360	273093360	**273093360**	*0*

Tabulka 14 zobrazuje výsledky měření úlohy 3 před optimalizací.

Tabulka 14 - Úloha 3 před optimalizací

ZTEST3	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
doba běhu [µs]	49012581	48921582	49471805	51242153	50181111	**49765846**	*964181*
využití paměti [kB]	357451800	357451800	357451800	357451800	357451800	**357451800**	*0*

Výsledky měření úlohy 4 jsou v tabulce 15.

Tabulka 15 - Úloha 4 před optimalizací

ZTEST4	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
doba běhu [µs]	47842876	49307073	48678786	48362543	49868032	**48811862**	*793609*
využití paměti [kB]	357451800	357451800	357451800	357451800	357451800	**357451800**	*0*

V následující tabulce 16 jsou výsledky úlohy 5 před optimalizací.

Tabulka 16 - Úloha 5 před optimalizací

ZTEST5	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
doba běhu [µs]	99744228	94258295	103634253	101519675	98048576	**99789113**	*4015466*
využití paměti [kB]	714902272	714902272	714902272	714902272	714902272	**714902272**	*0*

9.2 Výsledky měření po optimalizaci

Následující sada tabulce obsahuje výsledky měření po optimalizaci. Výsledky měření úlohy 1 jsou v tabulce 17.

Tabulka 17 - Úloha 1 po optimalizaci

ZTEST1_OUT	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
doba běhu [µs]	49569422	45548706	47012300	48079402	46840488	**47410064**	*1504836*
využití paměti [kB]	357451512	357451512	357451512	357451512	357451512	**357451512**	*0*

V tabulce 18 jsou výsledky měření úlohy 2 po optimalizaci.

Tabulka 18 - Úloha 2 po optimalizaci

ZTEST2_OUT	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
doba běhu [µs]	183625	176280	181568	183384	181550	**181214**	*3415*
využití paměti [kB]	7232608	7232608	7232608	7232608	7232608	**7232608**	*0*

Výsledky měření z úlohy 3 po optimalizaci jsou v následující tabulce 19.

Tabulka 19 - Úloha 3 po optimalizaci

ZTEST3_OUT	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
doba běhu [µs]	3925171	3935616	3995608	3991818	4109257	**3991494**	*73154*
využití paměti [kB]	357451944	357452160	357452160	357452160	357452160	**357452117**	*97*

Tabulka 20 obsahuje výsledky 4. úlohy po optimalizaci.

Tabulka 19 - Úloha 4 po optimalizaci

ZTEST4_OUT	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
doba běhu [µs]	2778469	2546984	2359725	2423280	2376976	**2497086.8**	*173499*
využití paměti [kB]	362689736	362689736	362689736	362689736	362689736	**362689736**	*0*

V tabulce 21 jsou výsledky 5 úlohy po optimalizaci.

Tabulka 20 - Úloha 5 po optimalizaci

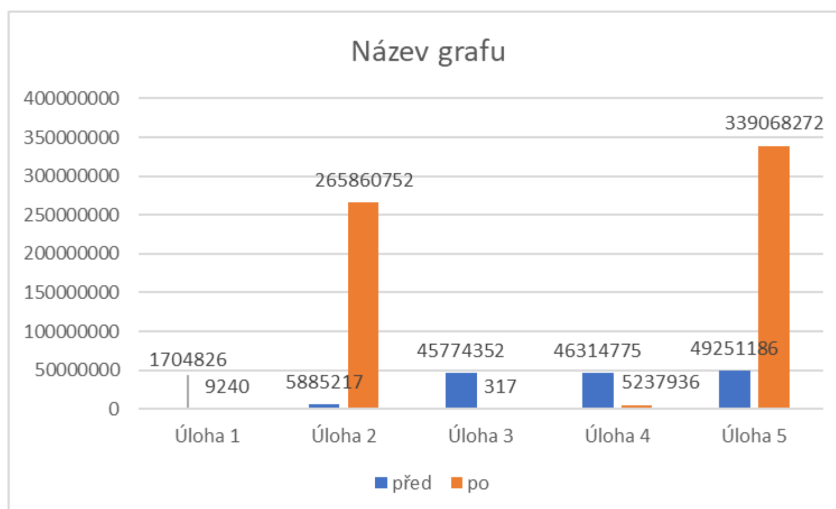
ZTEST5_OUT	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
doba běhu [µs]	50606245	48308721	52124432	51112307	50601184	**50537927**	*1614614*
využití paměti [kB]	375834000	375834000	375834000	375834000	375834000	**375834000**	*0*

V tabulce 22 jsou výsledky 6 úlohy po optimalizaci.

Tabulka 21 - Úloha 6 po optimalizaci

ZTEST6_OUT	měření 1	měření 2	měření 3	měření 4	měření 5	**PRŮMĚR**	*SMODCH*
doba běhu [µs]	143816586	127868893	139362726	156603475	146546371	**142839610**	*10492255*
využití paměti [kB]	714903600	714903600	714903600	714903600	714903600	**714903600**	*0*

Následující graf zobrazuje rozdíl mezi měřením před optimalizací a po optimalizaci.



Graf 3 - porovnání rozdílu měření před optimalizací a po optimalizaci

10 Diskuze k výsledkům optimalizace za použití optimalizátoru

V úloze 1, kde byly provedeny všechny tři optimalizace, algoritmus správně rozeznal systémové proměnné od uživatelských a provedl jejich optimalizaci. Při optimalizaci příkazů *select* a *readtable* optimalizátor vyhledal použité sloupce a použil je k optimalizaci. Došlo zde k ušetření paměti i zkrácení doby běhu

V úloze 2 algoritmus vyhodnotil správně, které proměnné jsou uživatelské a zároveň již neoptimalizoval proměnné, které mají požadovaný prefix. Při optimalizaci *select* došlo k výrazné změně alokace prostředků - přibližně 97 procent.

Ve 3. úloze se optimalizoval pouze příkaz *select*. Příkaz *readtable* se naopak neoptimalizoval. Došlo zde ke zlepšení doby běhu o jeden řád, tedy o 90 procent. Využití paměti ale zůstalo stejné.

Ve 4. úloze se provádí všechny optimalizace. Tato testovací úloha je zaměřena na kontrolu optimalizace příkazu *select.* Došlo ke 20násobnému zlepšení doby běhu a k částečnému zlepšení paměti.

V úloze 5 se testuje optimalizace *read*. Opakovaným měřením bylo potvrzeno zkrácení doby běhu a ušetření paměti až o polovinu.

Nejlepšího výsledku při použití optimalizátoru bylo dosaženo v úloze 6. Před optimalizací nebylo možné změřit dobu běhu a využití paměti. Doba běhu přesáhla maximální limity a server vždy ukončil běh programu. Avšak po optimalizaci bylo měření již úspěšné. Tímto bylo dokázáno, že optimalizace má zásadní vliv na plynulý chod aplikace.

Na obecné rovině bylo tedy prokázáno, že použitím optimalizátoru bylo dosaženo zlepšení doby běhu programu a úspory využité paměti.

Důležité je si uvědomit, že toto jsou výsledky pouze jednoduchých testovacích scénářů, které nám současná gramatika dovoluje vytvořit. Jak jsem zmiňoval, gramatika zatím neobsahuje například možnost použít v kódu funkce a jejich volání. V praxi se totiž často vyskytuje rekurze, případně cyklické volání funkcí, a v tu chvíli se úsporné prostředky, především paměť, značně zvyšují.

Závěr

Jedním z cílů této práce bylo změřit zkrácení doby běhu a zmenšení množství využité paměti pro vybrané struktury jazyka ABAP. Předpokladem byla existence míst ve zdrojovém kódu, která jsou vhodná k optimalizaci. Nejprve bylo provedeno 8 druhů měření nad testovacími daty o velikosti přibližně 1.3 milionu záznamů pro zjištění vhodných míst k optimalizaci. Měřením bylo zjištěno, že největší úspora doby běhu programu a využití paměti programem byla při čtení dat z databázové nebo interní tabulky.

V případě čtení dat z databázové tabulky se v 80 % vyskytoval příkaz „SELECT *“ a program tak zabíral zbytečně mnoho paměti a doba běhu programu byla výrazně delší. Výběrem pouze požadovaných sloupců bylo možné zkrátit dobu běhu programu až desetkrát a využití paměti až o 90 %. Tyto hodnoty byly přímo závislé na množství vybíraných sloupců a na datovém typu vybíraných polí.

Výkonost příkazu READ TABLE byla závislá na typu použité interní tabulky. Měřením bylo zjištěno, že ze tří typů interních tabulek (STANDARD, SORTED a HASHED) je nejčastěji využívaný typ STANDARD. Využitím tohoto typu tabulky bylo dosaženo dobrých výsledků při přidávání nových záznamů. Tento proces proběhl v řádu mikrosekund. Naopak při čtení z tohoto typu interní tabulky byly zjištěny velmi nedostatečné hodnoty. Vyhledávací algoritmus procházel jeden řádek po druhém a hledal požadovanou hodnotu klíče. Vzhledem k velikosti procházené tabulky se program zdržel průměrně o 50 ms. Přesto, že se tato hodnota může zdát zanedbatelná, vyskytuje se tento příkaz často v cyklech či rekurzích, což může vést k navýšení doby běhu programu na několik minut.

Měřením bylo zjištěno, že typy tabulek SORTED a HASHED jsou při čtení rychlejší o 4 řády. Avšak při vkládání nových tabulek bylo změřeno, že typ tabulky SORTED je pomalejší než zbylé dva typy. Nejrychlejší typ tabulky je ve všech operacích typ tabulky HASHED. Tímto měřením se potvrdil výše zmíněný předpoklad o existenci vhodných míst k optimalizaci a byla zjištěna výše ušetřených prostředků serveru.

Dalším cílem této práce bylo vytvořit nástroj pro automatickou optimalizaci – optimalizátor. Optimalizátor byl vytvořen z dílčích částí: převod zdrojového kódu na strom programu, optimalizaci stromu programu a převod stromu programu zpět na zdrojový kód.

Nejprve jsem implementoval gramatiku jazyka ABAP, kterou jsem poté použil k převodu zdrojového kódu na syntaktický strom. Po získání syntaktického stromu je strom převáděn na strom programu. To byla požadovaná struktura, která reprezentuje zdrojový kód při optimalizaci. Veškeré úlohy byly prováděny nad tímto stromem.

Optimalizace stromu programu byla v rámci diplomové práce realizována na třech operacích. První a druhá úloha byla optimalizace výkonu. Jednalo se o optimalizaci čtení dat z databáze a využití vhodného typu interní tabulky. Druhým využitím optimalizace byla optimalizace čitelnosti zdrojového kódu. Pro simulaci této funkčnosti byla použita poslední úloha. Tato úloha rozšiřuje jména uživatelských proměnných o definovaný prefix. Poslední částí optimalizátoru byl převod stromu programu zpět na zdrojový kód. Zde byla využita stromová struktura. Výhoda tohoto použití je v tom, že optimalizátor začne přepisovat na kód vrchol stromu a následně se rekurzivně zanořuje. Tím je docílen převod na zdrojový kód, který svým zanořením odpovídá původnímu kódu. Posledním cílem práce bylo testování funkčnosti optimalizátoru. Tato měření byla realizována na sadě testovacích úloh. Bylo dokázáno očekávané zrychlení a úspora paměti.

Bohužel samotné využití tohoto programu je limitováno gramatikou jazyka ABAP. V současné době tato gramatika není kompletní a nelze tak optimalizovat libovolný kód jazyka ABAP. Proto jako další směr rozvoje této práce se zaměřím na rozšíření této gramatiky za účelem plného využití tohoto nástroje.

Reference

[1] *What is ERP?*, 2020. Dostupné z: http://www.netsuite.com/portal/resource/articles/erp/what-is-erp.shtml

[2] PANG, A., M. MARKOVSKI a A. MICIK. Top 10 ERP Software Vendors, Market Size and Market Forecast 2018-2023. *Apps Run The World.* 2020. Dostupné z: https://www.appsruntheworld.com/top-10-erp-software-vendors-and-market-forecast/

[3] SVOBODOVA, L a M. CERNA. Accounting, Economic and ERP Systems on the Czech Scene. *Advanced Science Letters*. 2020. DOI:10.1166/asl.2016.6683

[4] *SAP SE*. 2020. Dostupné z: https://www.sap.com/

[5] KÜHNHAUSER, K. *ABAP: výukový kurz*. Brno: Computer Press, 2009. ISBN 978-80-251-2117-7.

[6] SAP ABAP. *Tutorialspoint*. 2020. Dostupné z: https://www.tutorialspoint.com/sap_abap/sap_abap_tutorial.pdf

[7] BANDARI, K. *Complete ABAP*. 1. vyd. B.m.: SAP Press, 2017. ISBN 978-1-4932-1273-6.

[8] Internal Tables in SAP ABAP. *Stechies*. Dostupné z: https://www.stechies.com/internal-tables/

[9] PROCHÁZKA, J. *Měření výkonosti vybraných struktur jazyka ABAP*. 2020

[10] PASAM, Phani Kumar. Performance optimization of SQL statements in ABAP. *SAP Blogs*. 2014. Dostupné z: https://blogs.sap.com/2010/08/24/performance-optimization-of-sql-statements-in-abap/

[11] ABAP Code Optimization Methods & Techniques. *SAP Brains Online*. Dostupné z: https://sapbrainsonline.com/abap-tutorial/abap-code-optimization-methods-techniques.html

[12] BOES, S. *The SQL Trace (ST05) - Quick and Easy*. 2007. Dostupné z: https://blogs.sap.com/2007/09/05/the-sql-trace-st05-quick-and-easy/

[13] GRUNE, D. *Modern Compiler Design*. New York: Springer Science+Business Media, 2012.

[14] MOGENSEN, Torben Ægidius. *Introduction to Compiler Design*. 1. London: Springer, 2011. ISBN 978-0-85729-829-4.

[15] WATSON, D. *A Practical Approach to Compiler Construction*. B.m.: Springer International Publishing AG, 2017. ISBN 978-3-319-52789-5.

[16] PARR, T. *The Definitive ANTLR Reference: Building Domain-Specific Language*. B.m.: Pragmatic Bookshelf, 2007. ISBN 978-0-9787392-5-6.

[17] TOMASSETTI, G. *The ANTLR Mega Tutorial*. Dostupné z: https://tomassetti.me/antlr-mega-tutorial/

Příloha A – obsah repositáře

Veřejný repositář na adrese https://github.com/starefire/abap_opt obsahuje veškeré zdrojové soubory optimalizátoru a sady testovacích souborů. Soubory jsou rozděleny do složek *grammars*, *src*, *tests, out*.

grammars – obsahuje původní gramatiku jazyka ABAP, novou gramatiku jazyka ABAP a gramatiku Optimalizátoru.

- abap.g4
- original_grammar.g4
- optimalizator.g4

src – obsahuje zdrojové soubory optimalizátoru. Tyto soubory jsou rozděleny do po

- AST
- META-INF
- Optimalizotor
- Main.java

Složka AST obsahuje složku *gen*, *Nodes* a soubor *BuildAstVisitor.java*. Složka gen obsahuje generované soubory z nástroje ANTLR. Tyto soubory je ovšem po generování nutné upravit, aby umožňovali pracovat s typem *InfixNode*. Složka Nodes obsahuje různé typy uzlů pro tvorbu stromu programu. Soubor *BuildAstVistor.java* obsahuje logiku pro tvorbu stromu programu. Využívá generované soubory ze složky gen a různé typy uzlů stromu ze složky Nodes.

Složka optimalizátor obsahuje složku *gen* a soubory *Optimalizator.java* a *setOptimizer.java*. Složka gen obsahuje generované soubory z nástroje ANTLR. Tyto soubory pak využívá třída *setOptimizer*, která nastavuje vnitřní proměnné singletonu *Optimalizator*. Ten je v souboru *Optimalizator.java*.

V souboru main.java dochází ke spojení všech částí dohromady. Vstupem optimalizátoru je soubor *test.txt*, výstupem je pak soubor *testout.txt.* Vstupní soubormusí být ve stejné složce jako optimalizátor, respektive jeho spustitelný soubor *ABAP_optimalizator.jar*, který je ve složce *out*.