Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Automatizace procesů v DBMS ORACLE

Bc. Jindřich Novotný

©2017 ČZU v Praze

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Jindřich Novotný

Informatika

Název práce

Automatizace procesů v DBMS ORACLE

Název anglicky

Automatization of processes in DBMS ORACLE

Cíle práce

Diplomová práce je zaměřena na problematiku automatizace procesů v databázových systémech ORACLE.
Hlavním cílem této práce je:
a) objasnit teoretické principy relačně databázové technologie se zřetelem na problematiku automatizace procesů v rámci provozovaných databází ORACLE,
b) zmapovat momentální stav této problematiky a vymezit její relevantnost včetně požadavků na ni kladených,
c) navrhnout přijatelné řešení této záležitosti v souladu s identifikovanými požadavky,
d) ověřit přínosnost navržených záležitostí na funkčním prototypu,
e) ověřené záležitosti zobecnit pro další možná uplatnění.

Metodika

Použitá metodika zadané diplomové práce bude založena na studiu a analýze dostupných informačních zdrojů a existujících řešení v dané oblasti. Stěžejními při vypracování této závěrečné práce budou techniky a postupy relačně databázové technologie. Navrhované řešení bude zohledňovat identifikované požadavky a očekávání spojená s řešenou záležitostí. Na podkladě syntézy teoretických poznatků a dosažených výsledků budou formulovány závěry této diplomové práce a následně zobecněny pro další možná použití.

Závazný harmonogram řešení zadané DP:

- Teoretické principy řešené problematiky, literární rešerše – předmět 1. zápočtu z DP: do 5.9.2016

- Zmapování momentální situace řešené problematiky, identifikace požadavků s tím spojených: do 20.12.2016

- Navržení možného řešení a jeho následné ověření formou praktického řešení – předmět 2. zápočtu z DP: do 28.2.2017

- Zobecnění navržených záležitostí pro další možná použití – předmět 3. zápočtu z DP: do 28.3.2017

Doporučený rozsah práce

55-65

Klíčová slova

Relačně db technologie, ORACLE, PL/SQL, triggery, funkce a procedury

---

Doporučené zdroje informací

BRYLA, B – LONEY, K. Mistrovství v Oracle Database 11g. Brno: Computer Press, 2010. ISBN 978-80-251-2189-4.

FEUERSTEIN, S. – PRIBYL, B. Oracle PL/SQL Programming, 6th Edition. O'Reilly Media, 2014. ISBN 978-1-4493-2445-2.

LACKO, Ľ. Oracle : správa, programování a použití databázového systému. Brno: Computer Press, 2007. ISBN 978-80-251-1490-2.

LONEY, K. Oracle Database, kompletní průvodce. Brno: Computer press, 2010. ISBN 978-80-251-2489-5.

PROCHÁZKA, D. Oracle : průvodce správou, využitím a programováním nad databázovým systémem. Praha : Grada, 2009. ISBN 978-80-247-2762-2.

URMAN, S. – HARDMAN, R. – MCLAUGHLIN, M. Oracle – Programování v PL/SQL. Brno: Computer Press, 2010. ISBN 978-80-251-1870-2.

VALENTA, M. – POKORNÝ, J. *Databázové systémy.* Praha: České vysoké učení technické v Praze, 2013. ISBN 978-80-01-05212-9.

Předběžný termín obhajoby

2016/17 LS – PEF

Vedoucí práce

doc. Dr. Ing. Václav Vostrovský

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 1. 11. 2016

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 1. 11. 2016

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 20. 02. 2017

Čestné prohlášení

Prohlašuji, že svou diplomovou práci „Automatizace procesů v DBMS ORACLE” jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 31.3.2017 ____________________________

Poděkování

Rád bych touto cestou poděkoval doc. Ing. Václavu Vostrovskému, Ph.D. za vedení mé diplomové práce, za veškeré odborné rady, jež mi poskytl a za čas, který mi věnoval během přípravy diplomové práce.

Automatizace procesů v DBMS ORACLE

Automatization of processes in DBMS ORACLE

Souhrn

Diplomová práce je zaměřena na problematiku automatizace procesů v databázových systémech Oracle a její využitelnost v rámci podnikatelských subjektů. Automatizace procesů v podnikových databázích se dnes stává běžnou praxí a není již výsadou pouze větších podniků. Ne vždy je však její potenciál plně využit a v některých případech je tato problematika i celkově přehlížena. Jedním z cílů této diplomové práce je tedy poukázat na důležitost dané problematiky, možnosti, jenž poskytuje a na výhody, které sebou přináší. V teoretické části autor vymezuje pojmy z relačně databázové technologie používané v kontextu s problematikou automatizace procesů v DBMS Oracle, a to zejména metody a techniky programovacího jazyka PL/SQL, triggery a joby. V praktické části je pak navržen funkční prototyp řešící problematiku monitoringu databázových událostí, na němž je názorně ukázáno využití popsaných principů v praxi.

Klíčová slova

Relačně databázové technologie, ORACLE, PL/SQL, automatizace, monitoring, triggery, funkce, procedury, joby, XML

Summary

This thesis is focused on the automatization of processes in the Oracle database systems and its usability in the context of business subjects. Automatization of processes in corporate databases is becoming a common practice and it is not only a privilege of larger companies. Its full potential is not always fully exploited and in some cases, the problematics is basically overlooked. One of the goals of this thesis is thus to highlight the importance of the problematics, opportunities which it provides and the advantages that it brings. In the theoretical part, the author defines the concepts of relational database technology used in the context of the problematics of process automatization in Oracle DBMS, particularly the methods and techniques of programming language PL/SQL, triggers and jobs. In the practical part is then designed a working prototype dealing with the issues of database events monitoring on which there is clearly demonstrated the use of described theoretical principles in practice.

Keywords

Relational database technology, ORACLE, PL/SQL, automatization, monitoring, triggers, functions, procedures, jobs, XML

Obsah

Seznam obrázků 4

Seznam tabulek 4

1 Úvod 5

2 Cíl práce a metodika 7

2.1 Cíl práce 7

2.2 Metodika 7

3 Relevantnost problematiky automatizace procesů 8

4 Teoreická východiska 11

4.1 Jazyk PL/SQL 11

4.1.1 struktura PL/SQL bloku 12

4.1.2 Proměnné 13

4.1.3 Řízení toku programu 16

4.1.4 Správa výjimek 22

4.1.5 Procedury, funkce a balíky 25

4.2 Triggery 31

4.2.1 DML triggery 31

4.2.2 DDL triggery 33

4.2.3 Systémové triggery 34

4.3 Joby 35

5 Praktická část **38**

5.1 Popis navrhovaného řešení 38

5.2 Návrh datových struktur 40

5.2.1 XSD snapshotu 40

5.2.2 Datový model 41

5.3 Tvorba balíku 44

5.3.1 Hlavička balíku 44

5.3.2 Pomocné procedury a funkce 46

5.3.3 Automatizace sběru dat 48

5.3.4 Tvorba snapshotů 53

5.3.5 Automatizace monitorovacího nástroje 56

5.3.6 Práce s datovým skladem 59

6 Diskuse a doporučení **65**

7 Závěr **66**

Literatura **68**

Přílohy **69**

Seznam obrázků

1 Implicitní konverze mezi datovými typy [7] 14
2 Struktura příkazu IF [zdroj: autor] 17
3 Struktura příkazu CASE [zdroj: autor] 17
4 Logické dělení cyklů [zdroj: autor] 20
5 Struktura jednoduché smyčky [zdroj: autor] 21
6 Struktura FOR cyklu [zdroj: autor] 21
7 Struktura WHILE cyklu [zdroj: autor] 22
8 Struktura EXCEPTION bloku [zdroj: autor] 23
9 Business schéma hlavních funkčních celků prototypu [zdroj: autor] 39
10 XSD schéma snapshotu [zdroj: autor] 42
11 Návrh pomocných tabulek pro měřené DB [zdroj: autor] 43
12 Návrh datového skladu [zdroj: autor] 45

Seznam tabulek

1 Příklady předdefinovaných výjimek v PL/SQL [zdroj: autor] 24
2 Aktivační události DDL triggerů [zdroj: autor] 33
3 Aktivační události systémových triggerů [zdroj: autor] 34

1 Úvod

Tématem této diplomové práce je automatizace procesů v databázových systémech Oracle a její využitelnost v rámci podnikatelských subjektů. Automatizace procesů v podnikových databázích se dnes stává běžnou praxí a není již výsadou pouze větších podniků, avšak ne vždy je její potenciál plně využit a v některých případech je tato problematika i celkově opomíjena. Daná problematika je však v dnešní době velmi důležitá zejména proto, že velké množství systémů používaných v podnikovém prostředí používá jako základ právě databázové systémy od společnosti Oracle. Díky automatizaci firemních procesů pak mohou společnosti soustředit své síly na hlavní podnikové činnosti, namísto aby byl jejich personál vytížen zpracováním rutinních činností, jež mohou být automatizovány. Toto činí z automatizace procesů v databázových systémech důležitou složku pro rozvoj podniku.

Automatizace v podnikových databázových systémech je pak využívána hned z několika důvodů. Prvním z důvodů je množství dat v podnikových systémech a rychlost jejich hromadění. Vzhledem k rostoucí velikosti datové základny, s níž se dnes většina podniků potýká, se stává manuální zpracování pro některé typy procesů velice neefektivním ba dokonce nerealizovatelným v reálném čase a při přijatelných nákladech. Dalším důvodem jsou pak přínosy plynoucí z automatizace procesů, jako příklady lze uvést zrychlení firemních procesů, snadná použitelnost a opakovatelnost procesů, zvýšení kontroly nad probíhajícími procesy či odstranění chyb způsobených působením lidského faktoru a mnohé další. Z těchto i z dalších důvodů je třeba se touto problematikou podrobněji zabývat. Nicméně i přesto však bývá vývoj v této oblasti některými firmami i dnes stále podceňován.

A kde je možné s problematikou automatizace procesů v databázových systémech setkat? Jelikož v rámci podnikových databázových systémů lze automatizovat velké množství různých aktivit, od vytváření objektů, přes monitorování a sběr dat, provádění rutinních úprav nad daty či automatizace reakcí na nastalé změny v databázi a mnohé další, s využitím automatizace lze realizovat také business logika. Využitelnost automatizace je tedy dostupná pro širokou škálu firemních procesů. Vzhledem k velkému rozsahu této problematiky se pak s její aplikací můžeme setkat nejen v malých společnostech, ale především pak ve středních a velkých podnicích, jenž mají často celé vývojové týmy zaměřené na vývoj nových funkcionalit pro automatizaci nejrůznějších procesů. Nejčastěji se pak s automatizací setkáme u podniků operujících v bankovním sektoru, jelikož tyto společnosti vždy pracují s obrovským množstvím dat, které již samo o sobě v podstatě znemožňuje provádění některých procesů manuálně. Avšak automatizace procesů v databázových systémech rozhodně není pouze výsadou velkých bank.

Tato práce je rozdělena do několika částí tak, aby co nejvíce obsáhla celou problematiku automatizace procesů v databázových systémech Oracle. V první části budou nejprve podrobněji rozebrány důvody, proč je nezbytné se zabývat problematikou automatizace procesů. Jaké výhody a nevýhody s sebou může tato záležitost přinášet pro podniky samotné a také, co je a není vhodné v podnikové databázi automatizovat. Následně ve druhé části diplomové práce budou objasněny základní principy programovacího jazyka PL/SQL, jenž jsou nezbytné pro vytváření funkcionalit, které jsou hlavním jádrem samotné automatizace procesů. Dále budou v této části rozebrány metody a techniky používané v rámci automatizace, konkrétně budou objasněny principy stojící za dvěma nejdůležitějšími nástroji používanými při automatizaci procesů, tedy konkrétně triggery a joby. Poslední část poté bude navazovat na teoretická východiska popsaná v předchozích částech a bude se zabývat tvorbu modelového balíku umožňujícího automatizaci monitoringu různých databázových událostí. V závěru pak budou shrnuty jednotlivé poznatky, jež vyvstanou v průběhu zpracování této diplomové práce.

Téma této diplomové práce jsem zvolil zejména proto, že se již delší dobu zabývám vývojem pro databázové systémy Oracle, a proto je mi téma týkající se automatizace procesů v DBMS Oracle blízké. Zkušenosti s touto problematikou jsem získal především jako zaměstnanec společnosti zabývající se poskytováním služeb v oblasti technologií Oracle.

2 Cíl práce a metodika

2.1 Cíl práce

Diplomová práce je zaměřena na problematiku automatizace procesů v databázových systémech Oracle a její využitelnost v rámci podnikatelských subjektů. Hlavním cílem závěrečné práce je navržení a realizace funkčního prototypu databázového balíku automatizujícího proces monitoringu vybraných databázových událostí, na němž budou demonstrovány možnosti praktického využití automatizace procesů v databázových systémech Oracle. Dílčími cíli závěrečné práce je zmapovat aktuální stav a vymezit relevantnost zvolené problematiky. Dále pak vytvořit literární rešerši obsahující úvod do zvolené problematiky, jenž bude zaměřena na objasnění důležitých principů relačně databázové technologie a jazyka PL/SQL v kontextu s problematikou automatizace procesů v rámci databázových systémů Oracle.

2.2 Metodika

Metodika zpracování této diplomové práce je založena na studiu a analýze dostupných informačních zdrojů a existujících řešení v dané oblasti. Hlavními informačními zdroji použitými v diplomové práci budou odborné publikace a oficiální dokumentace Oracle pojednávající o problematice relačně databázové technologie, především pak se zaměřením na automatizaci procesů a programování v jazyce PL/SQL. Stěžejní pro vypracování této závěrečné práce dále budou především metody a techniky relačně databázové technologie využívané v kontextu s problematikou automatizace procesů v DBMS Oracle v rámci podnikatelských subjektů, a to zejména programovací jazyk PL/SQL, triggery a joby. Navrhované řešení bude zohledňovat identifikované požadavky a očekávání spojená s řešenou záležitostí. Na podkladě syntézy teoretických poznatků a dosažených výsledků budou formulovány závěry této diplomové práce a následně zobecněny pro další možná využití.

1. V teoretické části budou při vytváření literární rešerše využity tištěné i elektronické informační zdroje, stejně tak jako praktické zkušenosti získané z firemní praxe a z předmětů absolvovaných během studia.

2. V praktické části budou nejprve použity programy Altova XMLSpy a Sybase PowerDesigner 15 pro navržení datových struktur, s nimiž bude navrhovaný balík pracovat. Poté bude využit program Oracle SQL Developer a databázový systém Oracle Database 12c, s jejichž pomocí bude v jazyce PL/SQL vytvořen a následně otestován funkční prototyp monitorovacího balíku.

3 Relevantnost problematiky automatizace procesů

Potřeba po automatizaci nejrůznějších lidských činností sahá hluboko do dějin lidstva. V průběhu času s postupným vývojem nových technologií se dále rozšiřovaly možnosti, co a do jaké míry mohlo být automatizováno. Tato diplomová práce se však bude zabývat výhradně metodami a technikami určenými k automatizaci procesů v rámci relačně databázových systémů Oracle. Především pak jazykem PL/SQL jakožto hlavním nástrojem používaným pro vytváření různých funkcionalit v rámci databáze, dále pak triggery a joby, jenž umožní automatizované spouštění připravených programů. Tato kapitola bude věnována relevantnosti problematiky automatizace procesů v DBMS Oracle, konkrétně zde bude popsáno, jaké výhody sebou automatizace přináší a také, co je a není vhodné v rámci databáze automatizovat.

Proč je důležité v rámci databází automatizovat a jaké výhody sebou automatizace přináší? Na tuto otázku neexistuje pouze jedna konkrétní odpověď, jelikož výhod, jenž sebou automatizace procesů přináší, je více. Jako příklady lze uvést některé z přínosů automatizace:

1. Zvýšení kontroly nad probíhajícími procesy. Díky automatizaci lze nejen určit, kdy jednotlivé činnosti proběhnou, ale také jejich vzájemná návaznost, díky tomu je možné zajistit, aby procesy probíhaly přesně tak, jak byly navrženy.

2. Snadná opakovatelnost prováděných činností. Ať již se jedná o jakýkoli proces, jestliže dojde k jeho automatizaci, je mnohem snazší provést daný úkon znovu. Každé další využití navíc zhodnocuje práci odvedenou na automatizaci onoho procesu, jelikož vývoj již nemusí být prováděn znovu, zatímco zpracování při manuálním provádění veškerých úkonů v rámci procesu by bylo nutné pokaždé opakovat.

3. Zrychlení procesů. Vzhledem k tomu, že v rámci automatizovaného procesu provádí veškerou práci databázový server, snad až na případné vstupy od uživatele, nezbytně dojde ke zrychlení celého procesu, jelikož co se výpočetní rychlosti týče mají počítače nespornou výhodu oproti člověku.

4. Zvýšení plynulosti procesů. Pokud je automatizovaný proces naprogramován správně, pak nedochází ke zbytečným prodlevám a čekání na určitá data, jako by tomu bylo v případě, že by stejnou práci prováděl lidský personál. Tento rozdíl je patrný zejména v případech, kdy generování určitých dat není primární náplní práce konkrétního zaměstnance, a tedy může dojít k výraznému zpoždění kvůli jiným úlohám. Oproti tomu automatizované úlohy mohou být plánovány na vteřiny přesně, a tedy vždy spuštěny v předem naplánovaný čas.

5. Odstranění chyb zapříčiněných lidským faktorem. Vzhledem k tomu, že vše provádí počítač, je zajištěno, že veškeré rozhodování proběhne vždy naprosto stejně, což eliminuje chyby z nepozornosti, jež se mohou projevit u lidí, zejména jedná-li se o dlouhodobou opakující se činnost, u níž může docházet ke ztrátě pozornosti.
6. Snadná použitelnost. Pokud jsou automatizované procesy navrženy správně, jejich nasazení či spuštění zvládne kdokoli bez toho, aniž by byl na daný úkol dlouze připravován a školen.
7. Usnadnění práce zaměstnancům. Pakliže bude zaměstnanci odebrána povinnost zabývat se rutinními úlohami, jež mohou být automatizovány, může se namísto toho věnovat důležitějším úkolům, díky čemuž se zvýší jeho pracovní efektivnost, jelikož není zatížen tolika činnostmi.
8. Možnost častěji nasazovat nové verze. Jestliže zautomatizujeme proces nasazení funkcionality, je možné dosáhnout při této činnosti časových úspor, díky čemuž je mnohem snazší nasazovat nové verze konkrétní funkcionality. Tento faktor umožňuje zkrácení intervalu mezi nasazením nových verzí funkcionalit, v nichž dochází k opravám chyb či rozšiřování funkčnosti.
9. Schopnost vyrovnat se s rostoucí datovou základnou. Jelikož množství dat, s nimiž dnes musí zejména velké společnosti pracovat, velice rychle narůstá, poskytuje automatizace procesů relativně snadnou cestu, jak se s tímto nárustem vyrovnat. Naopak při klasickém manuálním zpracování se po překročení určité hranice stává provedení některých činností praticky nemožné.

Oproti tomu nevýhody automatizace procesů de facto neexistují, jediné, co by se dalo považovat za nevýhodu, je nezbytnost mít v týmu zabývajícím se podnikovými databázemi specialistu na PL/SQL, v případě větších společností pak udržovat vývojový tým, či si tyto služby pronajmout externě. To sebou samozřejmě přináší jisté náklady a v případě externího dodavatele i jistá bezpečnostní rizika. Výhody plynoucí z automatizace procesů však tyto náklady snadno vyváží. Nicméně je nezbytné také zmínit, že ne veškeré podnikové procesy, je výhodné řešit přímo v databázi, například některé aplikace business logiky je výhodnější řešit pomocí enterprice aplikací, díky čemuž dosáhneme platformní nezávislosti.[3]

Nyní, když byla zodpovězena otázka přínosů automatizace, je důležité také zmínit, co je a co není v rámci databáze vhodné automatizovat. Je samozřejmé, že není vhodné pokoušet se o automatizaci procesů, v nichž významnou roli při rozhodování hraje lidský faktor, jelikož tuto složku rozhodování zatím není zcela možné nahradit. Dále není vhodné automatizovat procesy, v nichž dochází k vytváření či mazání indexů, popřípadě nastavování parametrů

serveru či databáze nebo procesy související s restartem databáze. Oproti tomu pro automatizaci jsou vhodné především opakující se úlohy, u nichž není třeba člověka, jenž by dělal rozhodnutí, popřípadě jedná-li se pouze o triviální rozhodnutí, která mohou být realizována pomocí podmínek v rámci větvení programu. Tato kategorie je velice široká, zmíníme tedy jen některé typy úloh, které je možno automatizovat. Jedná se například o úlohy typu sběr dat, vytváření objektů v databázi, periodické úpravy atributů v databázi či kontrolní procesy, jenž reagující okamžitě na nastalé změny, jako například na změny atributů, mazání, úprava nebo vytváření řádků a samozřejmě mnohé další.[6]

Závěrem této kapitoly lze tedy shrnout, že automatizace procesů je důležitým tématem v rámci práce s relačními databázemi, kterým je nezbytné se zabývat, ať již se jedná o malý či střední podnik, v rámci velkých podniků je pak automatizace procesů již téměř nezbytností. Přesto, že tato problematika má i jistá úskalí, výhody, jenž přináší, převažují veškeré potenciální náklady s ní spojené.

4 Teoreická východiska

4.1 Jazyk PL/SQL

Dříve, než je možné začít procesy automatizovat, je nejprve nutné naprogramovat řídící logiku podprogramů, které se stanou základem jednotlivých procesů, toto neplatí pouze pro databázové systémy Oracle či jiné relačně databázové systémy, ale obecně pro libovolné prostředí. V rámci databázových systémů Oracle existuje poměrně velké množství možností, jak vytvořit řídící logiku, v podstatě lze použít libovolný programovací jazyk, s jehož pomocí můžeme vytvořit aplikaci, jenž se dokáže připojit k databázi. Mezi ty nejvýznamnější a nejčastěji používané však patří jazyky z rodiny C, tedy C, C++ či C#, dále je třeba zmínit programovací jazyk Java, ten byl původně vyvinut společností Sun Microsystems, která však dnes již patří pod Oracle, a tudíž poskytuje velmi dobrou podporu pro jejich databázové systémy. Nejvýkonnějším a také nejpoužívanějším jazykem pro tvorbu podprogramů v systémech Oracle je však dnes programovací jazyk PL/SQL, a právě tím se bude zabývat tato část diplomové práce.[3]

PL/SQL neboli Procedural Language/Structured Query Language je procedurální programovací jazyk, který byl vytvořen s myšlenkou plné podpory databází a SQL. Tento programovací jazyk byl představen poprvé ve verzi Oracle 6, jenž byla vydána v červenci roku 1988. V době svého vzniku měl PL/SQL velmi omezené možnosti využití, nebylo například možné vytvářet procedury a funkce, ale bylo nutné pracovat s anonymními bloky kódu. Toto se však brzy změnilo díky neustálému vývoji a PL/SQL se tak postupně stalo nejpoužívanějším nástrojem pro vývoj řídící logiky podprogramů pracujících nad databázemi. K největším změnám v PL/SQL došlo především v Oracle 7, v této verzi byla přidána možnost pracovat s funkcemi, procedurami a balíky, dále pak v Oracle 9i, kdy se PL/SQL stalo plně objektově-relačním programovacím jazykem a nakonec ve verzi Oracle 11g, v níž byl jazyk transformován z původně interpretovaného jazyka na jazyk kompilovaný. V dalších verzích pak byly především odstraňovány dílčí nedostatky, přidávány nové funkcionality a také zvyšována rychlost výsledných programů. Dnes je podpora PL/SQL zabudována nativně i v dalších programovacích jazycích, jako jsou například Java či C# a tak je možné spouštět jednotlivé příkazy či volat PL/SQL procedury a funkce i z aplikací vytvořených v jiných programovacích jazycích. [3][6][7]

Díky čemu se stal jazyk PL/SQL nejvyužívanějším nástrojem pro vývoj podprogramů pracujících nad databázemi, když je na trhu velké množství dalších, již zmiňovaných programovacích jazyků, v nichž je možné dosáhnout stejného výsledku? Mezi některé z důvodů patří například tyto:

1. Jazyk PL/SQL umí pracovat přímo s datovými typy používanými v rámci SQL, a tak není třeba používat přetypování z jednoho typu na jiný, navíc díky možnostem deklarovat proměnné jako %TYPE či %ROWTYPE není nutné upravovat typy proměnných po každé změně v tabulkách, s nimiž v kódu pracujeme. Díky tomu lze programovat funkce a procedury, jež jsou mnohem více odolné vůči různým změnám v databázi.

2. Při tvorbě dotazů není nutné v rámci PL/SQL provádět operace otevření a uzavření dotazu, jako tomu je u jiných programovacích jazyků, tyto operace se zde provádí zcela automaticky.

3. Pokud vytváříme program, jenž má pracovat nad daty uloženými v relační databázi, pro dosažení stejného výsledku je při použití jazyka PL/SQL zapotřebí většinou výrazně menší množství kódu, navíc PL/SQL je přímo stavěn pro práci s databází, a tak jeho kompilátor dokáže odhalit i některé chyby, jež by se v jiných programovacích jazycích nemusely projevit a jejichž odhalení by poté bylo náročně.

4. V jazyce PL/SQL jsou přímo implementovány nástroje pro správu databáze, jako příklad lze uvést balíčky DBMS_STATUS či DBMS_RESOUCE_MANAGER a mnohé další.[3][6]

Tato část diplomové práce se bude zabývat jazykem PL/SQL a především pak způsoby, jakými je v tomto prostředí možné vytvářet řídící logiku, která je základem pro veškeré automatické procesy pracující s databázemi.

4.1.1 struktura PL/SQL bloku

Veškerý kód psaný v jazyce PL/SQL je zapisován do takzvaných bloků. V zásadě je možné rozlišovat dva typy bloků, anonymní bloky kódu a pojmenované bloky kódu, jako například funkce a procedury, které mají svá specifika, jež budou podrobněji popsána v části 4.1.5. Anonymní bloky kódu nejsou v rámci databáze ukládány a není možné je volat z jiného místa kódu ani z SQL dotazu. Oproti tomu funkce a procedury jsou bloky, které jsou v databázi uloženy pod svým unikátním jménem a je možné je volat v rámci PL/SQL kódu a funkce i v případě SQL dotazů. Každý blok kódu, ať již anonymní či pojmenovaný, má však vždy stejnou strukturu, ta sestává z deklarační části, jenž v případě anonymních bloků kódu začíná klíčovým slovem DECLARE, zde jsou deklarovány, popřípadě také inicializovány jednotlivé proměnné. Další částí je pak samotné tělo bloku začínajícího klíčovým slovem BEGIN a končící klíčovým slovem END, tato část je jako jediná povinná, jelikož se zde nachází konkrétní řídící logika, deklaraci proměnných a správu výjimek je možné v určitých případech

z PL/SQL bloku zcela vypustit. Poslední částí je již zmiňovaná část zpracovávající výjimky, ta je umístěna před klíčové slovo END v těle bloku a začíná klíčovým slovem EXCEPTION, v této části pak dochází k odchytávání a ošetření jednotlivých výjimek. Následující kód zobrazuje jak vypadá prototypická struktura PL/SQL bloku.[2][5][7][9]

```
[DECLARE]
   [deklarace proměnných]
BEGIN
   tělo bloku
[EXCEPTION]
   [zpracování výjimek]
END;
```

Jednotlivé bloky kódu lze do sebe navzájem zanořovat, tento postup je využíván například pokud je nutné ošetřit možné výjimky jen pro určitou část kódu, aniž by byl narušen další běh programu. V případě vytváření hierarchické struktury v rámci bloku kódu je však nezbytné, aby se navzájem nekřížili jednotlivé uvozující klauzule.[7]

4.1.2 Proměnné

Při vytváření programů je velmi často zapotřebí dočasně ukládat hodnoty, k tomuto účelu slouží v programovacích jazycích proměnné a PL/SQL není žádnou výjimkou. Jak již bylo řečeno v předchozí části proměnné je možné deklarovat jen a pouze v deklarační části bloku. V rámci PL/SQL rozlišujeme v zásadě dva typy proměnných, skalární a kompozitní proměnné.

4.1.2.1 Skalární proměnné

Skalární proměnné jsou ty, jenž v sobě uchovávají vždy pouze jedinou hodnotu. Jako skalární proměnné lze v PL/SQL použít veškeré datové typy dostupné v rámci SQL, ale také typy z nich odvozené. Mezi nejvyužívanější typy proměnných patří především číselné proměnné jako například NUMBER, znakové proměnné jako CHAR či VARCHAR2, dále pak datový typ DATE uchovávající datum nebo BOOLEAN, který je využíván především v rámci podmínek. Specifickými typy proměnných jsou pak takzvané „Large objects" neboli proměnné typu LOB, v nichž lze uchovávat nejrůznější objekty od textu až po obrázky či videa, do této kategorie patří například CLOB či BLOB. Alternativou k použití konkrétního datového typu je vytvoření takzvaně ukotvené proměnné za pomoci %TYPE, v tomto případě uvedeme název tabulky či pohledu a konkrétního sloupce, proměnná je pak ukotvena

k tomuto sloupci a sdílí s ním datový typ. Díky ukotvení je kód odolný vůči změně datového typu daného sloupce, problém nicméně může nastat při odstranění sloupce, na který se proměnná odkazuje.[3][4][7][8][12][15]

Mezi různými datovými typy skalárních proměnných lze v rámci kódu provést konverzi, některé z nich umožňují implicitní konverzi viz obrázek 1, u jiných je třeba provést explicitní konverzi pomocí funkce, takovýchto funkcí je velké množství, jako příklad lze uvést TO_CHAR, TO_NUMBER nebo TO_DATE. [5][7]

TO / FROM	BINARY_DOUBLE	BINARY_FLOAT	BINARY_INTEGER	BLOB	CHAR	CLOB	DATE	LONG	NCHAR	NCLOB	NUMBER	NVARCHAR2	PLS_INTEGER	RAW	UROWID	VARCHAR2
BINARY_DOUBLE		X	X		X			X	X		X	X	X			X
BINARY_FLOAT	X		X		X				X		X	X	X			X
BINARY_INTEGER	X	X			X				X		X	X	X			X
BLOB														X		
CHAR	X	X	X			X	X	X	X		X	X	X	X	X	X
CLOB					X				X			X				X
DATE					X			X	X			X				X
LONG					X		X		X			X		X		X
NCHAR	X	X	X		X	X	X	X		X	X	X	X	X	X	X
NCLOB					X	X		X	X			X				X
NUMBER	X	X	X		X			X	X			X	X			X
NVARCHAR2	X	X	X		X	X		X		X						X
PLS_INTEGER	X	X	X		X			X	X	X	X	X				X
RAW				X	X			X	X							X
UROWID					X				X	X		X				X
VARCHAR2	X	X	X		X	X	X	X	X	X	X	X	X	X	X	

Obrázek 1: Implicitní konverze mezi datovými typy [7]

V rámci deklarace proměnné je možné zároveň provést také inicializaci a přiřadit jí tak hodnotu ještě před začátkem těla bloku kódu. Skalární proměnné lze definovat jako klasické proměnné, u nichž je možné v průběhu programu měnit jejich hodnotu nebo jako konstanty, jenž musí být při definování zároveň inicializovány a jejich hodnotu již nelze v průběhu programu měnit. Pro vytvoření konstanty se při definici používá klíčové slovo CONSTANT. Následující ukázka kódu zobrazuje, jakým způsobem je možné provést deklaraci proměnné v PL/SQL.[5][7][12]

```
variable_name [CONSTANT] variable_type [:= literal_value];
variable_name [CONSTANT] table_name.column_name%TYPE [:= literal_value];
```

4.1.2.2 Kompozitní proměnné

Za kompozitní proměnné jsou považovány především záznamy a kolekce, jedná se tedy o proměnné, které v sobě uchovávají více dílčích hodnot, jež mohou být libovolného datového typu, který PL/SQL podporuje. Záznamy jsou kompozitní proměnné, jenž mohou obsahovat dílčí hodnoty různých datových typů, oproti tomu hodnoty obsažené v kolekci jsou vždy stejného datového typu.[7]

Záznamy je možné používat například namísto jednotlivých proměnných pro zachycení řádky vybrané z konkrétní tabulky či pohledu příkazem SELECT, popřípadě pro zachycení výstupu z kurzoru, kterým je věnována část 4.1.3.2. Při deklaraci záznamu je nutné stejně jako u skalárních proměnných definovat jeho typ, ten je buď možné explicitně nadefinovat nebo lze použít ukotvení, obdobně jako u skalárních proměnných, zde za pomoci klíčového slova %ROWTYPE. Následující ukázka znázorňuje, jakým způsobem je možné deklarovat záznam. Za předpokladu, že deklarovaný typ přesně kopíruje katalogový objekt, pak oba záznamy budou stejného typu. K dílčím hodnotám vytvořených záznamů poté přistupujeme pomocí tečkové notace, v níž zapíšeme název záznamu a název konkrétní dílčí hodnoty oddělené tečkou.[3][4][5][7][12]

```
TYPE record_type IS RECORD
(var1 variable_type[,var2 variable_type ,...]);
record1 record_type;
record2 catalog_object%ROWTYPE;
```

Dalším typem kompozitních proměnných jsou již zmíněné kolekce, ty můžeme v zásadě dělit na pole a listy. Pole jsou používána tam, kde přesně známe maximální počet položek, které chceme ukládat a v PL/SQL jsou reprezentována typem VARRAY. Listy naopak využijeme tam, kde nevíme kolik záznamů bude potřeba ukládat, v PL/SQL jsou listy reprezentovány takzvanými „nested tables", což je možné přeložit jako hnízděné tabulky, běžněji se však setkáme s původním označením. Před vytvořením kolekce je nutné vždy nejprve definovat její typ, opět pomocí příkazu TYPE, kolekci samotnou pak lze deklarovat buď prázdnou nebo je možné ji zároveň naplnit hodnotami. Ke konkrétní hodnotě v kolekci se dostaneme pomocí názvu kolekce a pozice dané hodnoty uzavřené v kulatých závorkách, oproti Javě či jazykům z rodiny C je zde však rozdíl v indexaci, která začíná od jedničky a ne od nuly, jak bývá většinou zvykem. Následující ukázka znázorňuje, jakým způsobem jsou definovány již zmiňované druhy kolekcí a definovány jejich typy, v těle kódu jsou pak za pomoci cyklů vypsány hodnoty všech prvků obou kolekcí.[7][12]

```
DECLARE
  TYPE varray_type IS VARRAY(10) OF data_type;
  array varray_type := varray_type(1,2,3,4,5,6,7,8,9,10);

  TYPE table_type IS TABLE OF data_type;
  list table_type := table_type(1,2,3,4,5,6,7,8,9,10);
BEGIN
  FOR i in 1..list.COUNT LOOP
    dbms_output.put('['||list(i)||']');
  END LOOP;

  FOR i in 1..array.LIMIT LOOP
    dbms_output.put('['||array(i)||']');
  END LOOP;
END;
```

4.1.3 Řízení toku programu

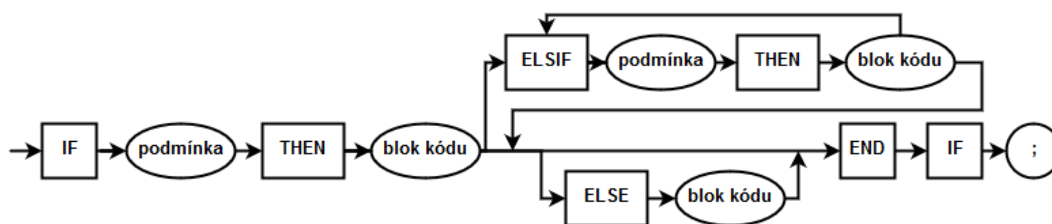
Tato část diplomové práce se bude zabývat řídícími strukturami používanými v rámci jazyka PL/SQL a to především jejich konstrukcí a funkcemi jenž zastávají. Řídící struktury jsou části kódu umožňující větvení kódu, opakování určitého chování či přístup k datům uloženým v databázi, a proto se s nimi setkáme téměř v každé netriviální proceduře či funkci.

4.1.3.1 Podmínková logika

Řídícím strukturám, v nichž určujeme, jaká část kódu bude provedena v závislosti na výsledku nějaké podmínky či hodnotě určité proměnné, se říká podmínková logika nebo také podmínky. V rámci PL/SQL rozlišujeme v zásadě dva typy podmínek, a to konkrétně IF a CASE, oba tyto typy podporují jak jednoduché, tak i vícenásobné větvení. Rozdíl mezi těmito dvěma strukturami je ve způsobu, jakým přistupují k aktu větvení kódu.

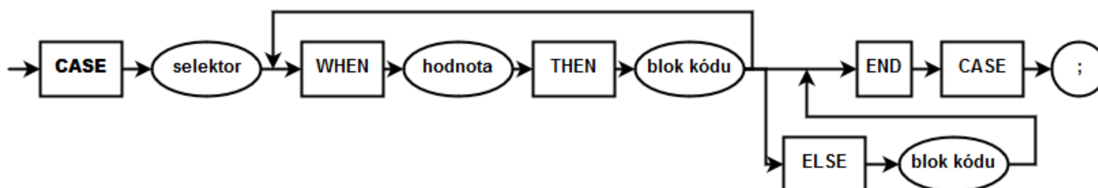
V rámci podmínkové logiky typu IF jde o větvení na základě podmínek, jenž mají booleovský výsledek. Podmínky mohou být jedno či více operandové, avšak v případě více operandových podmínek je třeba dbát na to, aby operandy byly stejných nebo navzájem převoditelných typů. V rámci podmínek lze využívat kromě klasických operátorů také funkce, a to buď jako jeden z operandů či samostatně, zde však musí být návratová hodnota typu BOOLEAN. Na obrázku 2 je vyobrazena struktura příkazu IF. Povinnou část příkazu tvoří vždy klíčové slovo IF následované podmínkou s booleovským výsledkem a klíčovým slovem THEN, další část tvoří blok kódu, jenž se provede v případě, že je podmínka splněna,

příkaz je poté zakončen klauzulí END IF a středníkem. Další, již nepovinné, části, kterými je možné rozšířit příkaz IF, jsou bloky ELSIF a ELSE, viz obrázek 2. Blok ELSIF umožňuje realizovat vícenásobné větvení přidáním dalších podmínek a podmíněných bloků kódu, zatímco ELSE umožňuje přidat blok kódu, jenž bude proveden, pokud žádná z předchozích podmínek nebyla splněna. V případě, že vytvoříme vícenásobné větvení, vždy se provede pouze ten blok kódu, který se nachází u první splněné podmínky, a to i přesto, že by byly splněny i další podmínky.[2][7][12]



Obrázek 2: Struktura příkazu IF [zdroj: autor]

Na rozdíl od rozhodovací struktury typu IF, u typu CASE je větvení prováděno na základě hodnoty konkrétní proměnné neboli selektoru a jako podmínka je vždy použit vztah rovnosti. Struktura příkazu CASE je znázorněna na obrázku 3. Povinná část příkazu je složena z klíčového slova CASE a rozhodující proměnné. Dále pak z klíčového slova WHEN, konkrétní hodnoty, jenž bude porovnávána se selektorem, klíčového slova THEN a bloku kódu, který bude proveden, pokud se selektor shoduje s danou hodnotou, jestliže tuto část zopakujeme s různými hodnotami, dosáhneme vícenásobného větvení. Povinná část je poté zakončena klauzulí END CASE a středníkem. Pro přidání bloku kódu, jenž se provede v případě, že selektor neodpovídá žádné z hodnot, je možné obdobně jako u IF i zde použít klíčové slovo ELSE následované blokem kódu. V případě, že je větvení založeno na podmínkách rovnosti, je vždy lepší použít příkaz CASE, jelikož dosáhneme zkrácení zdrojového kódu a jeho struktura je navíc přehlednější než u ekvivalentního příkazu IF.[2][7][12]



Obrázek 3: Struktura příkazu CASE [zdroj: autor]

4.1.3.2 Kurzory

Velmi často je nutné, aby konkrétní funkce či procedura pracující nad databází přistupovala k datům, pro tento účel jsou v jazyce PL/SQL určeny takzvané kurzory. Díky kurzorům je možné přímo v kódu používat různé SQL příkazy pro přístup k datům. Kurzory můžeme dělit například na základě počtu řádků dat, s nimiž pracují, rozeznáváme jedno řádkové neboli single-row a více řádkové neboli multi-row kurzory. V rámci jednořádkových kurzorů se využívají příkazy typu SELECT INTO, u nichž je nutné dohlédnout na to, aby výstupem byl pouze jediný řádek, v opačném případě by došlo k chybě ORA-01422, díky tomuto typu příkazů je možné z databáze uložit hodnoty do lokálních proměnných. V rámci více řádkových kurzorů jsou využívány klasické příkazy SELECT, zde však většinou vrací více řádků, a tak není možné je přímo vkládat do jedné proměnné, namísto toho jsou veli často využívány v cyklech. Kurzory je dále také možné dělit podle typu na implicitní a explicitní.[7]

Implicitní kurzory jsou automaticky vytvářené databázovým serverem, jedná se v podstatě o jakékoli SQL přímo zapsané v boku PL/SQL kódu. Výhodou implicitních kurzorů je jejich menší logistická složitost, není nutné je deklarovat v deklarační části bloku a databázový server je také automaticky otevírá a zavírá, díky čemuž jsou navíc robustnější vůči chybám. Nevýhodou však je, že není možné, aby programátor sám definoval vybírání dat z kurzoru, díky čemuž je použití implicitních kurzorů omezeno na single-row příkazy, multiple-row implicitní kurzor je možné vytvořit pouze ve spojení s cyklem typu FOR, jenž dokáže sám vybírat data z kurzoru. I přes tuto nevýhodu je však vhodné implicitní kurzory používat všude tam, kde je to možné.[7][9]

Explicitní kurzory jsou, na rozdíl od těch implicitních, definovány programátorem v deklarační části PL/SQL bloku, ručně je dále třeba řídit i vybírání řádek dat, včetně otevírání a zavírání kurzoru, jedinou výjimkou jsou již zmíněné FOR cykly, jenž tyto úkony provádí samy. Oproti implicitním kurzorům zde má programátor mnohem více volnosti v tom, jak s kurzorem pracovat a je tedy možné ho využít ve více situacích, nicméně jak již bylo řečeno, pokud to situace dovoluje, je lepší využít implicitní kurzory. Otevření explicitního kurzoru se provádí příkazem OPEN, uzavření pak příkazem CLOSE a pro vybírání dat je používán příkaz FETCH INTO, jednoduchý příklad znázorňující použití kurzoru demonstruje následující ukázka kódu, ve které je deklarován kurzor, z nějž je následně vybrána jedna řádka dat do před připraveného záznamu.[7][9]

```
DECLARE
  CURSOR cursor_name IS static_sql_select_statement;
  record_name cursor_name%ROWTYPE;
```

```
BEGIN
    OPEN cursor_name;
    FETCH cursor_name INTO record_name;
    CLOSE cursor_name;
END;
```

Kromě jednoduchých statických kurzorů lze vytvářet také dynamické kurzory, jenž umožňují v rámci použitého SQL příkazu využívat proměnné, které se mohou podle potřeb měnit vždy při otevírání kurzoru. Jestliže pracujeme s dynamickými kurzory, je třeba přidat za jméno kurzoru při jeho vytváření závorku obsahující jména a typy proměnných oddělené čárkami, ty následně mohou být využity v SQL příkazu, jenž je součástí kurzoru, dále pak při otevírání kurzoru musíme přidat závorku s atributy, které budou dosazeny do příslušných proměnných, obdobně jako při volání funkcí. Dynamické SQL má kromě své variability ovšem i stinné stránky, jako například menší rychlost, nemožnost kontrolovat platnost příkazu nebo ověřovat datové typy či velikosti při kompilaci kódu a další, proto by mělo být využíváno, jen pokud je to nezbytné.[7]

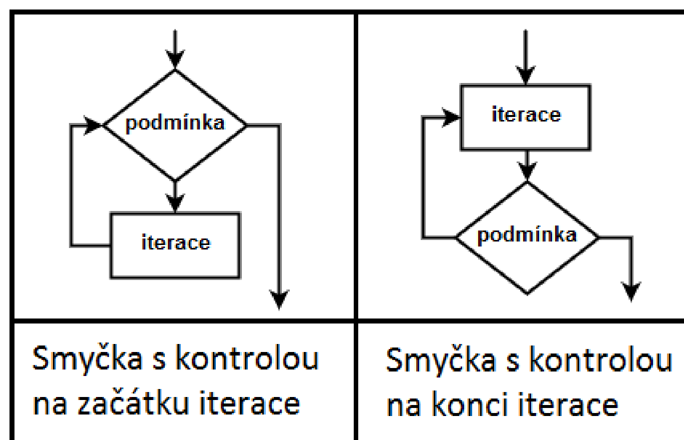
Pro práci s kurzory jsou definovány čtyři atributy, jež jsou využívány především v rámci podmínek, jedná se konkrétně o atributy %FOUND, %NOTFOUND, %ISOPEN a %ROWCOUNT, první tři atributy jsou booleovské, zatímco poslední z nich je typu NUMBER. Význam těchto atributů je následující:

- %FOUND určuje, zdali je možné zachytávat záznam z kurzoru, neboli jestli existuje další řádek, s nímž je možné pracovat, pokud takovýto řádek existuje, hodnota tohoto atributu je rovna TRUE. Společně s následujícím atributem je často využíván v podmínkách cyklů, které určují bude-li prováděn následující krok.
- %NOTFOUND je v podstatě negací předchozího atributu a nabývá hodnoty TRUE v případě, že již žádný další řádek neexistuje.
- %ISOPEN určuje, je-li kurzor otevřen či nikoli, tento atribut je často využíván v podmínkové logice obalující samotnou práci s kurzorem, díky tomu je možné předejít chybě při čtení ze zavřeného kurzoru.
- %ROWCOUNT vrací počet z kurzoru doposud zachycených řádků. Jako jediný z atributů je použitelný i pro implicitní kurzory a je využíván především pro kontrolu, zda-li byl zachycen alespoň jeden řádek.[7]

Pro ukládání dat z kurzoru je možné použít buď dostatečné množství skalárních proměnných nebo jednu kompozitní proměnnou příslušného typu, viz část 4.1.2, takovýto kurzor se v případě FOR cyklu vytvoří automaticky namísto klasické referenční proměnné.[7][9]

4.1.3.3 Cykly

Cykly neboli smyčky jsou řídící struktury, jenž dovolují v rámci programu opakovat určité chování tím, že spouští konkrétní část kódu tak dlouho, dokud platí podmínka pro pokračování smyčky. Cykly je možné rozdělit podle toho, zda provádí kontrolu podmínky na počátku iterace či na jejím konci, a tedy tělo cyklu vždy proběhne alespoň jednou, rozdíl mezi těmito typy graficky zobrazuje obrázek 4. Systémy Oracle však nativně podporují pouze cykly s kontrolou podmínky na počátku iterace, pakliže je třeba vytvořit cyklus s kontrolou na konci iterace, je nutné přizpůsobit k tomuto účelu takzvanou jednoduchou smyčku. V jazyce PL/SQL pak lze vytvářet v zásadě tři typy cyklů, konkrétně jde o jednoduché smyčky, dále pak o FOR a WHILE cykly.[2][4][7]



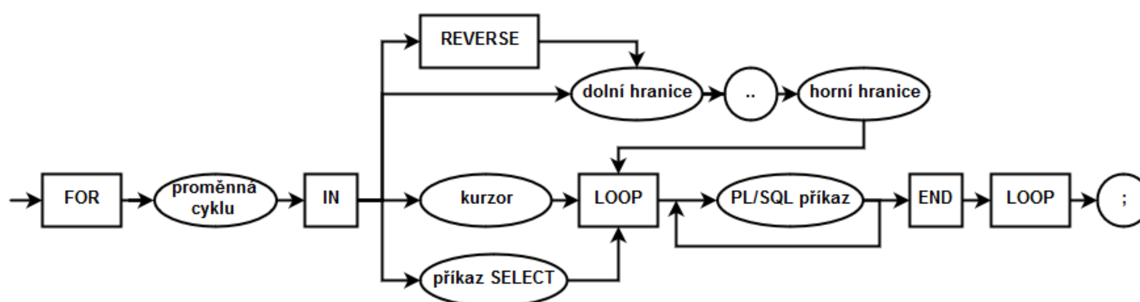
Obrázek 4: Logické dělení cyklů [zdroj: autor]

Jednoduché smyčky jsou nejjednodušší variantou cyklů, přesto jsou však často využívány tam, kde by nebylo nasazení cyklů FOR či WHILE vhodné kvůli jejich pevné pozici kontroly platnosti podmínky, ta se provede vždy alespoň jednou, a to na počátku iterace. Oproti FOR a WHILE cyklům nemá jednoduchá smyčka tento bod pevně stanoven, a tak lze vytvořit smyčky s kontrolou podmínky uprostřed či na konci bloku kódu uvnitř těla cyklu. Jednoduchá smyčka začíná klíčovým slovem LOOP a končí klauzulí END LOOP a středníkem, toto platí i pro ostatní typy smyček, v nich je však přítomna dále samotná podmínka cyklu. Aby nedošlo k vytvoření nekonečné smyčky, je nutné do těla cyklu umístit příkaz EXIT či EXIT WHEN. Příkaz EXIT ukončí cyklus okamžitě a většinou je používán společně s příkazem IF, aby došlo k ukončení pouze za určitých okolností, tento mechanismus má v sobě automaticky zabudován příkaz EXIT WHEN, jenž ukončí cyklus pouze v případě, že je splněna podmínka zapsána za klíčovým slovem WHEN.[2][7][12][15]

Dalším typem smyček jsou cykly typu FOR. Cykly tohoto typu jsou využívány tam, kde je počet iterací předem znám nebo je-li tento počet dán množstvím řádků dat, jenž získáme SQL

Obrázek 5: Struktura jednoduché smyčky [zdroj: autor]

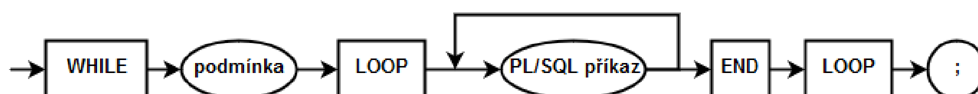
příkazem, popřípadě jako výstup z předem definovaného kurzoru, viz část 4.1.3.2. Výhodou cyklu typu FOR oproti ostatním typům smyček je především implicitní existence indexu iterací, který vzniká na počátku cyklu a zaniká po jeho ukončení. Index iterací může být různého typu, podle toho, jak je daný cyklus konstruován. Jestliže jde o klasický FOR cyklus s konkrétním rozsahem, bude index definován jako datový typ NUMBER, oproti tomu, pokud jde o cyklus s kurzorem či SQL příkazem, index bude automaticky definován jako %ROWTYPE pro daný výstup, v takovém případě pak k jednotlivým atributům přistupujeme v rámci cyklu za pomoci tečkové konvence. FOR cyklus začíná klíčovým slovem FOR a jménem indexu iterací pro danou smyčku, dále pak klíčovým slovem IN následovaným buď rozsahem nebo příkazem SELECT či kurzorem, dále je již FOR cyklus shodný s jednoduchou smyčkou. V případě FOR cyklu s rozsahem je možné použít také klíčové slovo REVERSE, v takovém případě pak dojde k procházení rozsahu v obráceném směru, tedy od horní hranice k dolní. Stejně jako u jednoduché smyčky je možné v těle cyklu použít příkazy EXIT či EXIT WHEN, avšak implicitně to není nutné, jelikož FOR cyklus sám hlídá podmínku ukončení smyčky na počátku každé iterace. Graficky je struktura FOR cyklů znázorněna na obrázku 6.[2][7][12][15]



Obrázek 6: Struktura FOR cyklu [zdroj: autor]

Posledním typem smyček jsou cykly typu WHILE, ty jsou používány tam, kde je potřeba opakovat určité chování po dobu platnosti konkrétní podmínky. Tento typ cyklu má stejně jako FOR cykly stanoven pevný bod kontroly platnosti podmínky pro pokračování smyčky na počátku každé iterace a stejně tak lze použít i aditivní příkazy EXIT či EXIT WHEN, ovšem na rozdíl od FOR cyklů nejsou smyčky typu WHILE založeny na rozsahu či výstupu z kurzoru, nýbrž na podmínkách booleovského typu. Dalším rozdílem je, že stejně

jako u jednoduchých smyček zde musí programátor sám řídit index iterací, pokud je zapotřebí. Od jednoduchých smyček se však, cykly typu WHILE neliší pouze konstrukcí, ale také tím, že proměnné v podmínce určující, zdali bude cyklus pokračovat musí být předem inicializovány, jinak program do smyčky nevstoupí, díky tomu není nutné provádět kontrolu je-li proměnná inicializována, na rozdíl od jednoduchých smyček, a tak se vyvarujeme zbytečným chybám. WHILE cykly začínají klíčovým slovem WHILE následovaným podmínkou booleovského typu a dále jsou již shodné s jednoduchými smyčkami. Graficky je struktura WHILE cyklů znázorněna na obrázku 7.[2][7][12]



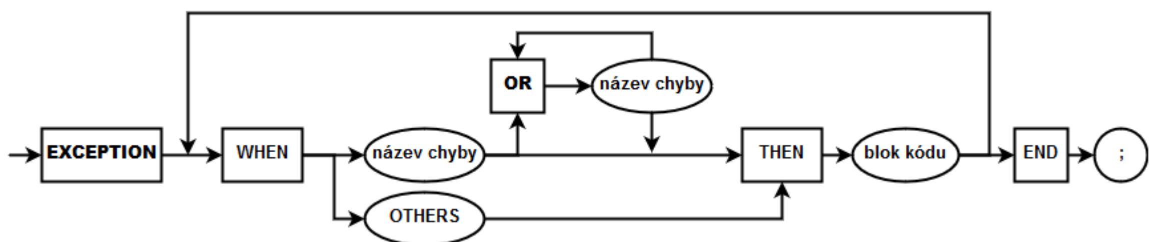
Obrázek 7: Struktura WHILE cyklu [zdroj: autor]

Od verze Oracle 11g se objevuje nový nástroj pro manipulaci se smyčkami, zmíněným nástrojem je příkaz CONTINUE a jeho alternativa CONTINUE WHEN. Díky těmto příkazům je možné manipulovat s během cyklu a přeskočit některé jeho iterace nebo jejich části, aniž by bylo nutné je složitě uzavírat do podmínkových bloků nebo dokonce smyčku přerušit. Příkazem CONTINUE docílíme okamžitého ukončení stávající iterace a přesunutí ovládání na počátek nové, tento příkaz je většinou používán v rámci podmínkového bloku, aby došlo k přeskočení iterace pouze za určité podmínky. Alternativou je použití příkazu CONTINUE WHEN, jenž ukončí iteraci jen tehdy, pokud je splněna podmínka zapsána na konci příkazu. Při použití těchto příkazů je však nutné pečlivě volit jejich umístění v těle cyklu, a to zejména u jednoduchých smyček a WHILE cyklů, jelikož u nich lze takto velmi snadno vytvořit nekonečné cykly, což je většinou nežádoucí stav. Oproti tomuto nebezpečí jsou imunní pouze cykly typu FOR, u nichž dochází k automatické inkrementaci či dekrementaci indexu iterací, avšak i zde je možné nevhodným umístěním příkazu CONTINUE zapříčinit nežádoucí chování programu.[7]

4.1.4 Správa výjimek

V každém programovacím jazyce je nutné mít nástroje pro ošetření nestandardních situací či chyb, v tomto ohledu není ani PL/SQL žádnou výjimkou. Jak již bylo řečeno v části 4.1.1, k odchytávání a ošetřování výjimek dochází v nepovinné části PL/SQL bloku, která je uvozena klíčovým slovem EXCEPTION a nachází se před koncem celého PL/SQL bloku. Tato část diplomové práce se bude zabývat způsobem, jakým jsou výjimky zpracovávány a také výjimkami samotnými.

Proces zpracování výjimky probíhá tak, že pokud při zpracování kódu dojde k nějaké chybě, ať již standardní či definované uživatelem, zastaví se další zpracovávání exekučního bloku a řízení bude předáno bloku pro správu výjimek. Následuje rozhodnutí o tom, jak bude výjimka zpracována, to probíhá obdobně jako v případě příkazu CASE, popsaného v části 4.1.3.1, tedy za pomoci klíčového slova WHEN, názvu chyby nebo klíčového slova OTHERS a klíčového slova THEN, pokud název chyby odpovídá řetězci v podmínkovém bloku, pak bude proveden příslušný blok kódu pro zpracování konkrétní výjimky. Jestliže je v podmínkové části klíčové slovo OTHERS, pak bude zachycena jakákoli výjimka, jenž nebyla dosud zachycena, jedná se tedy v podstatě o obdobu ELSE bloku. Jak vypadá struktura bloku pro správu výjimek znázorňuje graficky obrázek 8.[2][7][9][12][15]



Obrázek 8: Struktura EXCEPTION bloku [zdroj: autor]

V bloku kódu pro zpracování výjimek je možné používat dvě speciální proměnné, první z nich je SQLCODE, ta vrací číselný kód konkrétní chyby, až na jedinou výjimku je toto číslo vždy záporné, v případě NO_DATA_FOUND se však jedná o číslo sto. Pro uživatelem definované výjimky je určen rozsah -20001 až -20999, pokud uživatel nepřiřadí dané chybě číslo, bude však navrácena kladná hodnota jedna. Druhou ze zmiňovaných proměnných je SQLERRM, ta vrací označení a zprávu zachycené chyby. Poté co proběhne zpracování chyby dojde k ukončení procedury. Je však třeba zmínit, že v části kódu pro správu výjimek budou zachyceny pouze ty chyby, jež byly vyvolány v rámci exekuční části bloku. Chyby vyvolané při deklaraci či při samotném zpracování výjimky budou odchyceny v bloku kódu, z nějž byla příslušná funkce či procedura volána nebo v nadřazeném bloku kódu, pokud takový neexistuje, bude automaticky vyvolána neošetřená výjimka.[7][12]

Jak již bylo řečeno, v rámci jazyka PL/SQL je možné odchytávat jak předdefinované výjimky, tak i výjimky definované uživatelem. Předdefinované výjimky jsou v databázových systémech Oracle uloženy v balíku STANDARD, tyto standardizované výjimky se zaměřují především na obecné chybové stavy, nejčastěji používané výjimky jsou pojmenovány, nicméně větší množství výjimek nemá své jméno a je nutné je zachytávat v sekci OTHERS. V tabulce 1 jsou zapsány jednotlivé pojmenované výjimky včetně jejich kódů a příčiny vzniku.[2][7][12]

Tabulka 1: Příklady předdefinovaných výjimek v PL/SQL [zdroj: autor]

Název vyjímky	ORA kód	Příčina vzniku
ACCESS_INTO_NULL	ORA-06530	Přístup k neinicializovanému objektu
CASE_NOT_FOUND	ORA-06592	Žádná z podmínek v příkazu CASE neodpovídá selektoru a chybí klauzuje ELSE
COLLECTION_IS_NULL	ORA-06531	Přístup k neinicializovanému VARRAY nebo NESTED TABLE
CURSOR_ALREADY_OPEN	ORA-06511	Pokus o znovuotevření již otevřeného kurzoru
DUP_VAL_ON_INDEX	ORA-00001	Snaha o vložení duplicitní hodnoty do sloupečku s unikátním indexem
INVALID_CURSOR	ORA-01001	Provádění nepovolené operace nad kurzorem
INVALID_NUMBER	ORA-01722	Přiřazení nečíselné hodnoty do proměnné typu NUMBER
LOGIN_DENIED	ORA-01017	Pokus o log in s neplatnou kombinací jména či hesla
NO_DATA_FOUND	ORA-01403	Příkaz SELECT INTO nevrátí žádná data
NOT_LOGGED_ON	ORA-01012	Pokus o provádění operací nad DB bez navázaného spojení (například po ukončení session)
PROGRAM_ERROR	ORA-06501	Nalezen interní error PL/SQL
ROWTYPE_MISMATCH	ORA-06504	Nekompatibilní návratové typy kurzorů v úloze
SELF_IS_NULL	ORA-30625	Pokus o volání instance dynamického objektu, která dosud nebyla inicializována
STORAGE_ERROR	ORA-06500	SGA došla paměť nebo byla poškozena
SUBSCRIPT_BEYOND_COUNT	ORA-06533	Použití vyšší hodnoty indexu pro přístup k NESTED TABLE či VARRAY než kolik je v kolekci prvků
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	Použití nedovolené hodnoty indexu pro přístup k NESTED TABLE či VARRAY (nekladný integer)
SYS_INVALID_ROWID	ORA-01410	Poku o konverzi STRING na ROWTYPE, přičemž STRING nevyjadřuje platnou hodnotu ROWTYPE
TIMEOUT_ON_RESOURCE	ORA-00051	Databáze není schopna zajistit zámek na určité zdroje
TOO_MANY_ROWS	ORA-01422	Příkaz SELECT INTO vrátí více než jednu řádku
VALUE_ERROR	ORA-06502	Pokus o přiřazení příliž velké hodnoty do proměnné, která je příliš malá, než aby ji mohla udržet
ZERO_DIVIDE	ORA-01476	Pokus o dělené čísla nulou

Jestliže je nutné ošetřit nějaký nestandardní stav či chybu specifickou pro konkrétní program, je třeba definovat vlastní výjimku, toho je možné dosáhnout dvěma způsoby. Prvním způsobem je definování proměnné typu EXCEPTION přímo v deklarační části PL/SQL bloku, kterou je následně možné vyvolat v exekuční části bloku příkazem RAISE a názvem výjimky. Staticky definovaná výjimka má standardně hodnotu SQLCODE rovnu jedné a SQLERRM obsahuje text „user-defined exception", tento stav lze změnit pomocí direktivy PRAGMA EXCEPTION_INIT, s jejíž pomocí můžeme uživatelem definovanou výjimku asociovat s konkrétním číslem SQLCODE, v takovém případě pak SQLERRM obsahuje příslušnou zprávu dané chyby nebo prázdný řetězec, pokud chyba s daným číslem není definována.[7]

Druhou možností, jak vytvořit vlastní výjimku je její dynamické vyvolání, k tomuto se používá konkrétně příkaz RAISE_APPLICATION_ERROR, tento příkaz má dva parametry, prvním z nich je SQLCODE, jak již bylo řečeno výše, pro uživatelem definované výjimky je třeba dodržet rozsah -20001 až -20999, druhým parametrem je pak SQLERRM. Nevý-

hodou druhé varianty je, že jelikož jde o dynamicky vyvolanou výjimku, tak není v rámci bloku kódu pojmenována, a proto není možné ji v bloku pro správu výjimek odchytávat přímo za pomoci jména. Dynamicky vyvolané výjimky je možné zachytit v sekci OTHERS a následně rozlišit pomocí podmínky odkazující se na SQLCODE, alternativou je asociovat statickou výjimku pomocí direktivy PRAGMA, tento případ znázorňuje následující ukázka kódu.[7]

```
DECLARE
   jmeno_vyjimky EXCEPTION;
   PRAGMA EXCEPTION_INIT(jmeno_vyjimky, -20001);
BEGIN
   RAISE_APPLICATION_ERROR(-20001, 'zprava_vyjimky');
EXCEPTION
   WHEN jmeno_vyjimky THEN
      dbms_output.put_line(SQLERRM);
END;
```

4.1.5 Procedury, funkce a balíky

Doposud byly zmíněny pouze anonymními bloky kódu, ty jsou vhodné především ve chvíli, kdy jde o řešení jednorázových problémů. V praxi je však velmi často nutné určitou úlohu zpracovávat opakovaně, a právě pro tyto účely slouží procedury a funkce, ty umožňují ukládat bloky kódu přímo v databázi pod konkrétním jménem tak, aby bylo možné je v případě potřeby spouštět. Díky balíkům pak lze ze samostatných procedur a funkcí vytvořit komplexní celky řešící konkrétní problematiku.[7][9] Tato část diplomové práce se bude zabývat právě využitím a konstrukcí procedur, funkcí a balíků.

Využití procedur a funkcí přináší mnoho výhod, možnost vytvářet procedury a funkce s parametry přináší znovu použitelnost kódu pro různá zadání bez nutnosti zasahovat do kódu. Přemístěním kódu na databázový server pak dojde ke zvýšení výkonu v důsledku snížení přenášených dat mezi serverem a klientem, jenž dostává pouze výsledek dané operace. Další výhodou, avšak nikoli poslední, je zvýšení výkonu při opakovaném spouštění, díky tomu, že analýzy provedené při prvním průchodu daného kódu jsou uloženy v SGA (System Global Area) paměti a je možné je znovu použít při dalších spuštění programu. Při použití procedur, funkcí a balíků se navíc mnohem lépe kontroluje, kdo má k jakým funkcionalitám přístup, a to díky přidělování oprávnění konkrétním skupinám uživatelů či jednotlivcům.[7]

Pro spouštění funkce či procedury je nutné mít oprávnění EXECUTE pro spouštění konkrétní funkcionality či EXECUTE ANY, jenž umožňuje spouštět libovolnou funkcionalitu. Přidě-

lením oprávnění pro spouštění balíku získá uživatel právo spouštět libovolnou proceduru či funkci v něm uloženou. Obdobně pro vytváření procedur, funkcí a balíků je zapotřebí mít oprávnění, a to buď CREATE PROCEDURE pro tvorbu ve vlastním schématu v databázi, anebo CREATE ANY PROCEDURE, jenž umožňuje vytvářet procedury, funkce či balíky i ve schématech ostatních uživatelů. Při vytváření procedur, funkcí a balíků je však dále zapotřebí mít příslušná oprávnění ke všem databázovým objektům, s nimiž se v daném kódu pracuje, toto pravidlo lze obejít pouze nastavením autentizace při vytváření funkcionality z tvůrce (definer), což je implicitní hodnota, na uživatele používajícího funkcionalitu (current_user), tento postup bude podrobněji popsán v části 4.5.1.[2][5][7]

Funkce a procedury je možné spouštět pomocí několika typů notací, přičemž všechny obsahují jméno balíku, pokud se v nějakém nachází, a jméno funkce či procedury oddělené tečkou, dále pak závorku obsahující konkrétní hodnoty či proměnné reprezentující jednotlivé parametry oddělené čárkou. Rozdíl mezi jednotlivými typy notací je však v tom, jakým způsobem jsou zapisovány parametry v závorce za jménem funkce. Při použití klasické poziční notace je nutné zapisovat veškeré parametry v takovém pořadí, v jakém jsou definovány v hlavičce funkce či procedury, a tedy nelze vynechat ani parametry s definovanou implicitní hodnotou. Druhou notací je takzvaná jmenná notace, ve které je vždy zapsáno jméno parametru, operátor „=>” a konkrétní hodnota, ve jmenné notaci je možné uspořádat parametry v libovolném pořadí a ty, jenž mají definovánu implicitní hodnotu je možné zcela vynechat. Poslední možností je kombinace předchozích dvou, pro tuto notaci však musí platit, že veškeré nepojmenované parametry se musí objevit na začátku a přesně v takovém pořadí, v němž jsou uvedeny v hlavičce funkce či procedury.[7] Následující ukázka kódu znázorňuje použití jednotlivých notací při volání funkce se třemi parametry *a*, *b* a *c* typu NUMBER, jenž by vrátila vždy stejný výsledek.

```
begin
   jmeno_baliku.jmeno_funkce(3, 4, 5);
   jmeno_baliku.jmeno_funkce(c => 5, a => 3, => 4);
   jmeno_baliku.jmeno_funkce(3, c => 5, b => 4);
end;
```

4.1.5.1 Procedury

Procedury jsou typem podprogramů, jenž implicitně nemá žádnou návratovou hodnotu, a proto není možné je použít jako operand na pravé straně pro přiřazení hodnoty proměnné, v podmínkách či v SQL dotazech. Nejčastější využití procedur spočívá v provádění DML

operací, tedy vkládání, úprava či mazání dat a jsou velmi často volány za pomoci triggerů či jobů, s nimiž se blíže seznámíme v kapitole 5.[5][7][15]

Svou konstrukcí jsou procedury podobné anonymním blokům PL/SQL kódu, jediný rozdíl je, že namísto klíčového slova DECLARE mají na počátku hlavičku procedury. Hlavička procedury začíná klíčovými slovy CREATE PROCEDURE, mezi ty lze doplnit nepovinnou klauzuli OR REPLACE, díky níž dojde k nahrazení původní procedury se stejným jménem, pakliže taková v databázi již existuje. Dále následuje název samotné procedury a závorka obsahující definice jednotlivých parametrů oddělené čárkou, ta však není povinná v případě, že se jedná o proceduru bez parametrů. Každý parametr je definován jménem, módem a datovým typem.[2][5][15] Mód proměnné může být nastaven na následující tři hodnoty:

- IN - Jedná se o defaultní mód parametru, proměnné v tomto módu jsou určeny pouze pro čtení a jejich hodnotu nelze v průběhu funkce či procedury měnit. Jestliže je parametr v tomto módu, je možné mu přiřazovat defaultní hodnotu.
- OUT - Tento mód udává, že daný parametr je určen pouze pro zápis, to umožňuje vracet hodnoty i v rámci procedur, což standardně nelze, v případě funkcí pak přidat další proměnnou jejíž hodnota bude vrácena na místo volání. Takovémuto parametru však nelze přiřadit defaultní hodnotu, jinak bude při kompilaci kódu vyvolána výjimka PLS-00230. Dalším omezením je, že při volání podprogramu nelze za takovýto parametr dosadit literál, což by zabránilo zápisu do proměnné, vždy je tedy nutné použít proměnnou, v opačném případě bude vyvolána výjimka ORA-06577 v SQL z nějž spouštíme podprogram a PLS-00363 přímo v PL/SQL kódu.
- IN OUT - Tento mód je kombinací předchozích dvou a určuje, že daný parametr je možné nejen číst, ale také do něj zapisovat, platí pro něj však stejná omezení jako pro mód OUT, tedy nelze přiřadit defaultní hodnotu parametru a při volání podprogramu nelze použít literál, ale pouze proměnnou.[2][7][12]

Kromě jména, módu a datového typu lze k definici parametrů přidat klíčové slovo NOCOPY, jenž zajistí rychlejší předávání parametrů. V případě, že je parametr v módu IN, lze také definovat jeho defaultní hodnotu, a to za pomoci klíčového slova DEFAULT a konkrétní hodnoty, jenž bude použita v případě, že při volání procedury nebude parametr specifikován. Za definicí proměnných může následovat nepovinná část určující mód autentizace, jenž byl zmíněn již dříve, ten je uvozen klíčovým slovem AUTHID a jednou z hodnot DEFINER, která je defaultní hodnotou a určuje autentizaci podle práv tvůrce procedury či CURRENT_USER, jenž určuje autentizaci podle práv původce volání procedury. Hlavička procedury je pak

ukončena klíčovým slovem IS nebo AS. Následující kód znázorňuje syntaxi příkazu CREATE PROCEDURE.[2][7][12]

```
CREATE [OR REPLACE] PROCEDURE jmeno_procedury
[(parametr1 [IN | OUT | IN OUT] [NOCOPY] datovy_typ [DEFAULT hodnota])
[,...]]
[AUTHID {CURRENT_USER | DEFINER}]{IS | AS}
  [declaration_statements]
BEGIN
  execution_statements
[EXCEPTION]
  [exception_handling_statements]
END;
```

4.1.5.2 Funkce

Funkce jsou dalším typem podprogramů, oproti procedurám se liší především v tom, že vždy vrací hodnotu předem specifikovaného typu, díky této skutečnosti jsou velmi často používány například v podmínkách, pro přiřazování hodnot proměnným či například v SQL dotazech, tedy všude tam kde procedury použít nelze. Funkce jsou tedy mnohem variabilnější nežli procedury, obecně je však zažito pravidlo, že procedury jsou používány pro databázové akce, zatímco použití funkcí se omezuje spíše na různé výpočty.[5][7]

Vytváření funkcí je velmi podobné jako v případě procedur, nicméně u funkcí se vyskytuje několik specifik, na které se zde zaměříme především. Samotný příkaz opět začíná hlavičkou funkce, ta je složena z klíčových slov CREATE FUNCTION, mezi něž opět lze přidat klauzuli OR REPLACE, dále pak závorka se specifikací parametru, má-li funkce nějaké, hlavička je pak zakončena klíčovým slovem RETURN a datovým typem, jenž stanový typ návratové hodnoty funkce.[2][7][12]

Stejně jako u procedur, i zde lze opět nastavit některé nepovinné parametry, jedním z nich je opět AUTHID nastavující mód autentizace, viz 4.1.5.1, další jsou ovšem specifické pouze pro funkce. Je-li při vytvářen funkce použito klíčové slovo DETERMINISTIC, označujeme tak funkci za deterministickou neboli takovou, jenž při stejných vstupních hodnotách vrací vždy stejný výsledek. Toto je nezbytné, aby bylo možné použít funkci například v materializovaných pohledech či u indexů založených na funkci, pokud však takto označíme nedeterministickou funkci, pak při jejím použití na zmíněných místech bude vyvolána chyba ORA-30553. Dalším parametrem je RESULT_CACHE, jenž umožňuje zvýšit efektivitu funkce tak, že

ukládá její výsledky pro konkrétní hodnoty parametrů do SGA paměti a při následném volání se stejnými parametry již pouze vrací hodnoty, namísto provádění funkce. K parametru RESULT_CACHE lze přidat klauzuli RELIES ON následovanou jmény tabulek oddělených čárkami, tato klauzule způsobí mazání výsledků založených na datech z uvedených tabulek v případě, že dojde k nějaké změně v daných tabulkách.[2][7][12] Následující kód znázorňuje syntaxi příkazu CREATE FUNCTION.

```
CREATE [OR REPLACE] FUNCTION jmeno_funkce
[(parametr1 [IN | OUT | IN OUT] [NOCOPY] datovy_typ [DEFAULT hodnota])
[,...]]
RETURN datovy_typ
[AUTHID {CURRENT_USER | DEFINER}]
[DETERMINISTIC]
[RESULT_CACHE [RELIES ON (jmeno_tabulky1[,jmeno_tabulky2,...])]]
{IS | AS}
  [declaration_statements]
BEGIN
  execution_statements
[EXCEPTION]
  [exception_handling_statements]
END;
```

4.1.5.3 Balíky

Balíky jsou ve své podstatě jakýmisi knihovnami, v nichž mohou být seskupovány různé funkce, procedury, proměnné a definice specifických typů či výjimek, jež jsou nezbytné pro řešení konkrétní problematiky. Výhody použití balíků spočívají nejen v seskupení veškerých komponent nezbytných pro řešení problematiky do jednoho celku, který je kompaktnější kupříkladu při přenášení do jiné databáze, ale také se snáze spravují oprávnění nezbytná k jeho využití, jak již bylo řečeno výše, udělením práv k balíku má uživatel přístup k celému obsahu. V rámci balíku lze také používat mechanismus zvaný přetěžování (overloading) funkcí a procedur, díky tomu lze uvnitř balíku vytvářet více podprogramů se stejným názvem, ale rozdílnou signaturou, při volání již systém sám rozpozná, o jaký konkrétní podprogram jde v závislosti na množství a typech parametrů. Další výhodou balíků je možnost rozlišovat mezi veřejnými (public) a soukromými (private) komponenty, což zajistí, že uživatelé nebudou mít přístup ke komponentám, jenž mají být používány pouze jinými objekty uvnitř balíku.[4][5][7][9]

Konstrukce balíku probíhá odděleně ve dvou krocích, nejprve je nutné vytvořit jeho specifikaci a až poté lze vytvořit samotné tělo balíku. Specifikace balíku je vytvořena příkazem CREATE PACKAGE následovaným jménem balíku. Dále může být opět nastaven mód autentizace pomocí AUTHID, viz 4.1.5.1, v tomto případě však nastavíme mód globálně pro celý balík a v jednotlivých funkcích či procedurách jej již nelze měnit, v opačném případě bude vyvolána chyba PLS-00157. Nakonec opět následuje klíčové slovo IS nebo AS a samotná specifikace veřejných komponent, k nimž chceme uživatelům povolit přístup, tedy deklarace globálních proměnných, výjimek či typů a samozřejmě také hlavičky veřejných funkcí a procedur. Celý příkaz je jako obvykle zakončen klíčovým slovem END a středníkem.[2][5][7][12] Následující ukázka kódu znázorňuje syntaxi příkazu CREATE PACKAGE.

```
CREATE [OR REPLACE] PACKAGE jmeno_baliku
[AUTHID {DEFINER | CURRENT_USER}] {IS | AS}
  specifikace komponent
END;
```

Dalším krokem je vytvoření těla balíku, k tomuto účelu slouží příkaz CREATE PACKAGE BODY následovaný příslušným jménem balíku a klíčovým slovem IS nebo AS. Samotné tělo balíku pak obsahuje bloky a specifikace všech veřejných komponent definovaných ve specifikaci balíku, může také obsahovat další objekty, které jsou již privátní a k nimž uživatelé nemají přístup, takovéto objekty pak mohou být používány pouze v rámci daného balíku. Celý příkaz je opět zakončen klíčovým slovem END a středníkem.[2][5][7][12] Následující ukázka kódu znázorňuje syntaxi příkazu CREATE PACKAGE BODY.

```
CREATE [OR REPLACE] PACKAGE BODY jmeno_baliku {IS | AS}
  tělo balíku
END;
```

V případech, kdy balík obsahuje například business logiku, jenž má zůstat obchodním tajemstvím a není žádoucí, aby ji mohli uživatelé prohlížet nebo jí měnit, se pro skrytí PL/SQL kódu používá takzvaný balič (wrapper), díky kterému lze jednotlivé podprogramy spouštět, ale nelze číst jejich zdrojový kód ani ho měnit. Tento postup lze využít pro libovolný soubor s PL/SQL kódem, tedy i pro anonymní bloky kódu, nicméně v praxi je nejčastěji využíván právě na úrovni balíků.[5][7]

Nyní, když byl definován způsob, jakým je možné vytvářet podprogramy v rámci balíků, funkcí a procedur, je nutné vytvořit objekty, jenž umožní jejich automatické spouštění, čímž dojde v rámci databáze k vytvoření automatických procesů, ty mohou zajišťovat zpracování

rutinních úkonů či provádění nejrůznějších kontrol a usnadnit tak uživatelům jejich práci. V rámci databázových systému Oracle k těmto účelům slouží především triggery a joby, kterými se budou zabývat následující části diplomové práce.

4.2 Triggery

Triggery, neboli automatické spouště, jsou specifické programy obdobně jako například funkce či procedury, na rozdíl od nich však není možné triggery spouštět přímo, ale jejich spuštění je vždy závislé na proběhnutí konkrétní události, pro kterou byly vytvořeny.[1][8] Oproti funkcím, procedurám či balíkům navíc podléhají triggery následujícím omezením:

- Na rozdíl od funkcí, procedur či balíků, jenž mají neomezenou délku, tělo triggeru může mít nanejvýš třicet dva tisíce sedm set šedesát bajtů.
- V rámci těla triggeru nelze používat příkazy pro řízení transakcí (TCL), tedy COMMIT, ROLBACK či SAVEPOINT, ani příkazy pro kontrolu dat (DCL), tedy GRANT a REVOKE.
- PL/SQL kód obsažený v těle triggeru nelze skrýt za pomoci wrapování tak, jako u funkcí, procedur a balíků.
- V těle triggeru nelze definovat proměnné typu LONG či LONG RAW a proměnné *:new* a *:old* s těmito typy také nemohou pracovat.[7]

Kvůli těmto omezením je dobré, aby byl kód triggeru pokud možno co nejkratší a veškeré složitější úkony či výpočty by měly být prováděny pomocí volání funkcí či procedur. Díky schopnosti triggerů automaticky se spouštět při konkrétní události mají široké využití při automatizaci procesů v rámci databáze. Mezi nejčastější využití patří například kontrola a vynucování referenční integrity, kontrola chování DDL a DML příkazů, automatické doplňování primárních klíčů či vytváření logů o přístupech a chování uživatelů. Triggery lze v zásadě rozdělit na několik základních typů, DDL, DML a systémové triggery, každý z těchto typů má své využití a určitá konstrukční specifika, jenž budou objasněna dále.[2][7]

4.2.1 DML triggery

DML triggery jsou nejčastěji používaným druhem triggerů, spouštěčem zde jsou příkazy určené pro manipulaci dat, tedy INSERT, DELETE nebo UPDATE, v posledním případě

lze také určit, upravením, kterého sloupce má dojít k aktivaci triggeru. DML triggery jsou vždy svázány s konkrétní tabulkou, nad kterou je kontrolován výskyt dané události, v případě odstranění tabulky dojde k zneplatnění triggeru.[2][7]

Při vytváření DML triggeru je použit příkaz CREATE TRIGGER následovaný jménem triggeru, dále musí být použito klíčové slovo určující časování triggeru, zde jsou konkrétně tři možnosti, a to BEFORE, při které dojde ke spuštění před samotnou událostí, AFTER spouštějící trigger po konkrétní události a INSTED OF, tato možnost aktivuje trigger namísto dané události. Dále je třeba zadat konkrétní DML událost, při které se bude trigger spouštět, jednotlivé události lze navíc zřetězit tak, že je oddělíme pomocí klíčového slova OR, v takovém případě lze v rámci těla triggeru použít v podmínkách klíčová slova INSERTING, UPDATING a DELETING, jenž vracejí hodnotu TRUE, pokud byl trigger aktivován příslušnou událostí. Povinná část příkazu pak končí názvem tabulky, nad kterou bude trigger vytvořen. Mezi nepovinné části pak u DML triggerů patří klauzule FOR EACH ROW, pokud je uvedena, dojde k aktivaci triggeru zvlášť pro každý řádek tabulky, jenž je ovlivněn daným DML příkazem, v opačném případě bude trigger aktivován pouze jednou pro danou událost. Další nepovinnou částí je pak podmínková klauzule uvozená klíčovým slovem WHEN a konkrétní podmínkou, zde lze specifikovat dodatečnou podmínku, jenž musí být splněna, aby došlo k aktivaci triggeru. Příkaz je zakončen samotným tělem triggeru, jenž má stejnou konstrukci jako klasický anonymní blok PL/SQL kódu, vztahují se na něj však omezení uvedená výše.[2][6][7][14] Následující ukázka kódu znázorňuje syntaxi příkazu CREATE TRIGGER pro DML triggery.

```
CREATE [OR REPLACE] TRIGGER jmeno_triggeru
{BEFORE | AFTER | INSTED OF}
{INSERT | UPDATE | UPDATE OF sloupec1[,sloupec2[,sloupecN]] | DELETE}
ON jmeno_tabulky
[FOR EACH ROW]
[WHEN (podminka)]
tělo triggeru
```

Pokud se jedná o řádkový trigger, tedy při vytváření byla použita klauzule FOR EACH ROW, je možné v rámci těla triggeru přistupovat k pseudo záznamům *new a old,* a to za pomoci proměnných *:new* a *:old* s použitím tečkové konvence a názvu atributu, pokud se k těmto proměnným pokusíme přistupovat u triggerů, jenž nejsou řádkové, bude vyhozena výjimka PLS-04082. Obsahem proměnné *:new* je nový řádek, vytvořený v rámci DML příkazu, vyskytuje se pouze v případě UPDATE a INSERT, pokud je danou DML událostí DELETE, pak je v proměnné *:new* hodnota *null*. Obdobně proměnná *:old* obsahuje řádek, jenž má být

smazán či upraven a vyskytuje se pouze u událostí UPDATE a DELETE, v případě události INSERT je hodnota této proměnné *null*.[2][7][14]

Klasické DML triggery jsou často používány pro kontrolu chování DML příkazů či vytváření logů o aktivitě uživatelů nad určitou tabulkou, v případě řádkových DML triggerů jde pak například o automatické zadávání klíčových hodnot za pomoci sekvencí nebo kontrola a vynucování referenční integrity.[7]

4.2.2 DDL triggery

Dalším typem triggerů jsou takzvané DDL triggery, jejich aktivátorem se mohou stát nejrůznější události, jenž vytváří, mění či odstraňují objekty v rámci databáze, tedy události spouštěné příkazy pro definici dat. Seznam událostí, jenž mohou být spouštěčem DDL triggerů je zobrazen v tabulce 2. DDL triggery jsou velmi často využívány pro vytváření logů o různých činnostech uživatelů nebo například pro kontrolu důležitých schémat, aby nedošlo k odstranění nezbytných objektů či přidělení oprávnění.[2][7]

Tabulka 2: Aktivační události DDL triggerů [zdroj: autor]

DDL událost	Podmínka aktivace triggeru
ALTER	K aktivaci triggeru dojde, při modifikaci databázového objektu, ne však při změně databáze (ALTER DATABASE)
ANALYZE	K aktivaci triggeru dojde, pokud databáze sbírá nebo odstraňuje statistiky či validuje strukturu databázového objektu
ASSOCIATE STATISTICS	K aktivaci triggeru dojde, pokud databáze asociuje typ statistiky s databázovým objektem
AUDIT	K aktivaci triggeru dojde, při zahájení auditu pomocí příkazu AUDIT
COMMENT	K aktivaci triggeru dojde, pokud je do datového slovníku přidán záznam o databázovém objektu
CREATE	K aktivaci triggeru dojde, pokud je pomocí příkazu CREATE vytvořen nový databázový objekt, nevztahuje se na příkaz CREATE DATABASE
DISASSOCIATE STATISTICS	K aktivaci triggeru dojde, pokud databáze odebere typ statistiky databázovému objektu
DROP	K aktivaci triggeru dojde, pokud je pomocí příkazu DROP odebrán objekt z databáze
GRANT	K aktivaci triggeru dojde, po přidělení oprávnění příkazem GRANT
NOAUDIT	K aktivaci triggeru dojde, při zastavení auditu pomocí příkazu NOAUDIT
RENAME	K aktivaci triggeru dojde, při přejmenování databázového objetu příkazem RENAME
REVOKE	K aktivaci triggeru dojde, po odebrání oprávnění příkazem REVOKE
TRUNCATE	K aktivaci triggeru dojde, po odebrání řádků z tabulky pomocí příkazu TRUNCATE
DDL	K aktivaci triggeru dojde, pokud nastane kterákoli z výše uvedených DDL událostí

Stejně jako při vytváření DML triggerů, i v případě DDL triggerů je použit příkaz CREAT TRIGGER, jeho podoba je však změněna. Prvním rozdílem jsou samozřejmě události, jež mohou být spouštěčem DDL triggeru, viz tabulka 2, na rozdíl od DML triggerů navíc zde nedochází ke spojení s konkrétní tabulkou, DDL triggery jsou navázány na databázi jako

celek nebo na konkrétní schéma v databázi, a proto se za klíčovým slovem ON musí objevit klíčové slovo DATABASE nebo SCHEMA. Pokud se jedná o triggery na úrovni databáze, pak dojde ke spuštění, jestliže kterýkoli uživatel iniciuje konkrétní událost, pro kterou byl trigger vytvořen. Oproti tomu triggery na úrovni schématu budou aktivovány pouze dojde-li k dané události ve specifikovaném schématu, není-li před klíčovým slovem specifikováno žádné schéma, pak dojde k automaticky k navázání na schéma tvůrce triggeru.[2][6][7][14] Následující ukázka kódu znázorňuje syntaxi příkazu CREATE TRIGGER pro DDL triggery.

```
CREATE [OR REPLACE] TRIGGER jmeno_triggeru
{BEFORE | AFTER | INSTED OF} ddl_udalost ON
{DATABASE | [jmeno_schematu.]SCHEMA}
[WHEN (podminka)]
tělo triggeru
```

4.2.3 Systémové triggery

Posledním typem jsou takzvané systémové triggery, tento typ je aktivován některou z databázových událostí, jejichž seznam je zobrazen v tabulce 3. Systémové triggery jsou nejčastěji využívány pro monitorování a vytváření logů o přístupech uživatelů k databázi či sledování aktivity samotné databáze.[7]

Tabulka 3: Aktivační události systémových triggerů [zdroj: autor]

Databázová událost	Podmínka aktivace triggeru
SERVERERROR	K aktivaci triggeru dojde, pokud server vyhodí výjimku, kromě následujících případů ORA-01403, ORA-01422, ORA-01423, ORA-01034 a ORA-04030
LOGON	K aktivaci triggeru dojde, pokud dojde k přihlášení uživatele k databázi
LOGOFF	K aktivaci triggeru dojde, pokud dojde k odhlášení uživatele z databáze
STARTUP	K aktivaci triggeru dojde, pokud dojde ke spuštění databáze
SHUTDOWN	K aktivaci triggeru dojde, pokud dojde k vypnutí databáze
SUSPEND	K aktivaci triggeru dojde, pokud se transakce stane suspendovanou
DB_ROLE_CHANGE	K aktivaci triggeru dojde, při změně role ze "standby" do "primary" či naopak

Příkaz pro vytvoření systémového triggeru je podobný jako v případě DDL triggerů, má však svá specifika. První změnou jsou samozřejmě události, jež mohou být použity pro aktivaci triggeru, viz tabulka 3. Další změnou je počet možností časování triggeru, oproti ostatním typům, zde nelze použít INSTED OF, ale pouze BEFORE nebo AFTER, časování systémových triggerů navíc podléhá dalším omezením. Má-li být trigger spuštěn až po určité události, tedy je použito časování AFTER, lze jako spouštěcí událost použít pouze LOGON, STARTUP, SERVERERROR, SUSPEND nebo DB_ROLE_CHANGE. Naopak pro triggery spouštěné

před událostí s časováním BEFORE mohou být použity pouze události LOGOFF a SHUTDOWN. Dalším omezením je pak to, že AFTER STARTUP a BEFORE SHUTDOWN triggery lze použít pouze na úrovni databáze a nikoli schématu.[7] Následující ukázka kódu znázorňuje syntaxi příkazu CREATE TRIGGER pro systémové triggery.

```
CREATE [OR REPLACE] TRIGGER jmeno_triggeru
{BEFORE | AFTER} DB_udalost ON
{DATABASE | [jmeno_schematu.]SCHEMA}
tělo triggeru
```

4.3 Joby

Dalším mechanismem, jenž je schopen autonomně spouštět podprogramy, a tedy automatizovat procesy v rámci databáze, jsou takzvané joby. Job je objekt, který je v databázi reprezentován kolekcí metadat popisujících uživatelem definovanou úlohu, jenž má být jednou či opakovaně spuštěna, tato metadata jsou v systémech Oracle uložena jako jeden záznam v tabulce ALL_SCHEDULER_JOBS. Joby, obdobně jako triggery, nelze spouštět přímo, ale k jejich aktivaci dochází za pomoci takzvaného plánovače (scheduler), jenž průběžně spouští jednotlivé joby, tedy vytváří jejich instance, na základě nastaveného plánu vycházejícího z metadat jednotlivých jobů. Jedinou výjimku z tohoto pravidla tvoří procedura RUN_JOB, jenž umožňuje okamžité spuštění konkrétního jobu. [11]

Joby lze využít zejména v situacích, kdy je nutné provádět opakovaně nějakou rutinní činnost, díky použití jobů pak již není třeba takovýmito činnostmi zatěžovat personál, jenž může namísto toho vykonávat důležitější úlohy. Další využití jobů spočívá v plánování času spouštění podprogramů, jenž pro svou realizaci zabírají velké množství zdrojů a jsou časově náročné, tyto úlohy mohou být za pomoci jobů spouštěny automaticky v časech, kdy se předpokládá, že server nebude příliš vytížen. [13]

Pro práci s joby byl v systémech Oracle od verze 7 určen balík DBMS_JOB, ten je i nadále přítomen kvůli zpětné kompatibilitě, avšak od verze 10g byl nahrazen modernějším balíkem DBMS_SCHEDULER, ten má, na rozdíl od svého předchůdce, větší rozsah funkcí a je na základě předchozích zkušeností mnohem lépe odladěn. Při práci s joby jsou v rámci balíku DBMS_SCHEDULER důležité především procedury CREATE_JOB, DROP_JOB, SET_JOB_ARGUMENT_VALUE, SET_ATTRIBUTE, ENABLE a DISABLE, kterými se budeme v této části diplomové práci zabývat.[10][13]

Joby jsou vytvářeny voláním procedury CREATE_JOB, v rámci této procedury je třeba specifikovat následující parametry:

- job_name - Tento parametr specifikuje jméno jobu, to musí být unikátní v rámci jmenného prostoru (namespace). Jedná se o povinný parametr a pokud není specifikován, dojde k vyvolání výjimky. Unikátní jméno jobu však lze vygenerovat pomocí funkce GENERATE_JOB_NAME.

- job_type - Tento parametr specifikuje konkrétní typ jobu, opět jde o povinný parametr a jeho absence vyvolá výjimku. Při vytváření jobu lze zvolit ze čtyř typů, prvním z nich je PLSQL_BLOCK, v tomto případě bude job spouštět anonymní blok kódu a je tedy nutné, aby počet argumentů byl roven nule. Druhým typem je STORED_PROCEDURE, tento typ spustí proceduru uloženou v databázi, nikdy však nelze použít funkci či proceduru s parametry typu OUT či IN OUT, viz část 4.1.5.1. Třetím typem je EXECUTABLE, jedná se o externí job, jenž umožňuje spustit vše, co lze spustit z příkazové řádky operačního systému. Čtvrtým a posledním typem je CHAIN, tento typ jobů se odkazuje na řetězec, namísto jediné procedury a díky tomu může spustit více než jeden podprogram.

- job_action - Tento parametr specifikuje konkrétní akci, jenž bude po spuštění jobu provedena, obsah tohoto parametru se různí v závislosti na typu jobu. Pro PLSQL_BLOCK je to blok PL/SQL kódu. V případě STORED_PROCEDURE jde o jméno konkrétní procedury, jenž má být spuštěna. U typu EXECUTABLE je nutné specifikovat jméno externího spustitelného souboru včetně cesty k němu a argumentů pro příkazový řádek. A nakonec pro CHAIN je to jméno objektu typu CHAIN. V případě, že tento parametr nebude specifikován, dojde k vyvolání vyjímky při vytváření jobu. Je důležité zmínit, že tento parametr je textového typu a je třeba dát pozor na to, aby byl jeho obsah korektně uvozen jednoduchými uvozovkami, toto je nutné kontrolovat zejména u PL/SQL kódu.

- number_of_arguments - Tento parametr specifikuje počet argumentů, které bude job očekávat. Počet argumentů může být v rozsahu nula až dvě stě padesát pět, defaultní hodnotou tohoto parametru je nula.

- start_date - Tento parametr specifikuje datum a čas prvního spuštění, plánovač (sheduler) však nezaručuje spuštění přesně v daný okamžik, jelikož systém může být zahlcen a v takovém případě dojde ke spuštění až po uvolnění zdrojů.

- queue_spec - Tímto parametrem lze specifikovat frontu, do které budou zařazeny události aktivované tímto jobem.

- repeat_interval - Tímto parametrem lze specifikovat jak často bude job spouštěn, v případě, že není uveden, dojde k aktivaci pouze jednou.

- schedule_name - Tento parametr specifikuje jméno plánu, s nímž má být daný job asociován
- end_date - Tento parametr představuje datum po kterém job expiruje a nebude již dále spouštěn, end_date musí být pozdější datum než start_date, jinak dojde k vyhození vyjímky po povolení jobu.
- job_priority - Tímto parametrem lze nastavit relativní prioritu jobu oproti ostatním jobům dané třídy, pokud mají být ve stejnou chvíli spuštěny dva joby, bude spuštěn ten s vyšší prioritou. Parametr může nabývat hodnoty od jedné do pěti, přičemž jednička reprezentuje nejvyšší prioritu, defaultně je pak tento parametr nastaven na číslo tři.
- comments - Díky tomuto parametru lze v databázi k metadatům o jobu uložit také komentář, zde bývá často zapsána funkce daného jobu.
- enabled - Tento booleovský parametr specifikuje, zda bude job vytvořen povolený (TRUE) či nikolo (FALSE). V případě, že se jedná o job typu STORED_PROCEDURE s alespoň jedním atributem, pak je nutné vytvořit job tak, aby nebyl povolen, aby bylo možné přiřadit parametry před jeho aktivací.
- auto_drop - Tímto booleovským parametrem nastavíme, zda bude job smazán poté, co dojde k jeho dokončení.[10]

Poté, co je job úspěšně vytvořen, pokud je hodnota parametru number_of_arguments větší než nula, je nutné nastavit jednotlivé atributy pro volanou proceduru, toho docílíme za pomoci procedury SET_JOB_ARGUMENT_VALUE, kterou je třeba spustit pro každý argument zvlášť. Tato procedura má tři parametry, jméno jobu (job_name), pozici parametru (argument_position) a hodnotu parametru (argument_value). Poté, co jsou nastaveny veškeré atributy volané procedury, je možné povolit spouštění jobu, aniž by došlo k chybě.[10][13]

V případě potřeby lze jednotlivé parametry jobu dodatečně změnit, za tímto účelem je používána procedura SET_ATTRIBUTE, ta má tři parametry, a to jméno jobu (name), jméno atributu (attribute) a hodnotu daného parametru (value). Mezi další procedury pracující s joby patří již zmíněné DROP_JOB, ENABLE a DISABLE, všechny tyto funkce mají jako parametr jméno jobu, DROP_JOB a DISABLE mají navíc parametr force, jehož defaultní hodnotou je false. Procedura DROP_JOB umožňuje smazání konkrétního jobu, a to i pokud je daná job právě aktivní, v takovém případě je však nutné nastavit parametr force na hodnotu true. Procedura ENABLE pak daný job povoluje, zatímco procedura DISABLE jej zakazuje, neboli nastavují atribut enabled na hodnoty TRUE či FALSE, v případě procedury DISABLE pak lze ovlivnit, zda bude dokončen běh aktuální instance jobu, nastavením parametru na hodnotu true, či nikoli, jestliže ponecháme defaultní hodnotu false.[10][11][13]

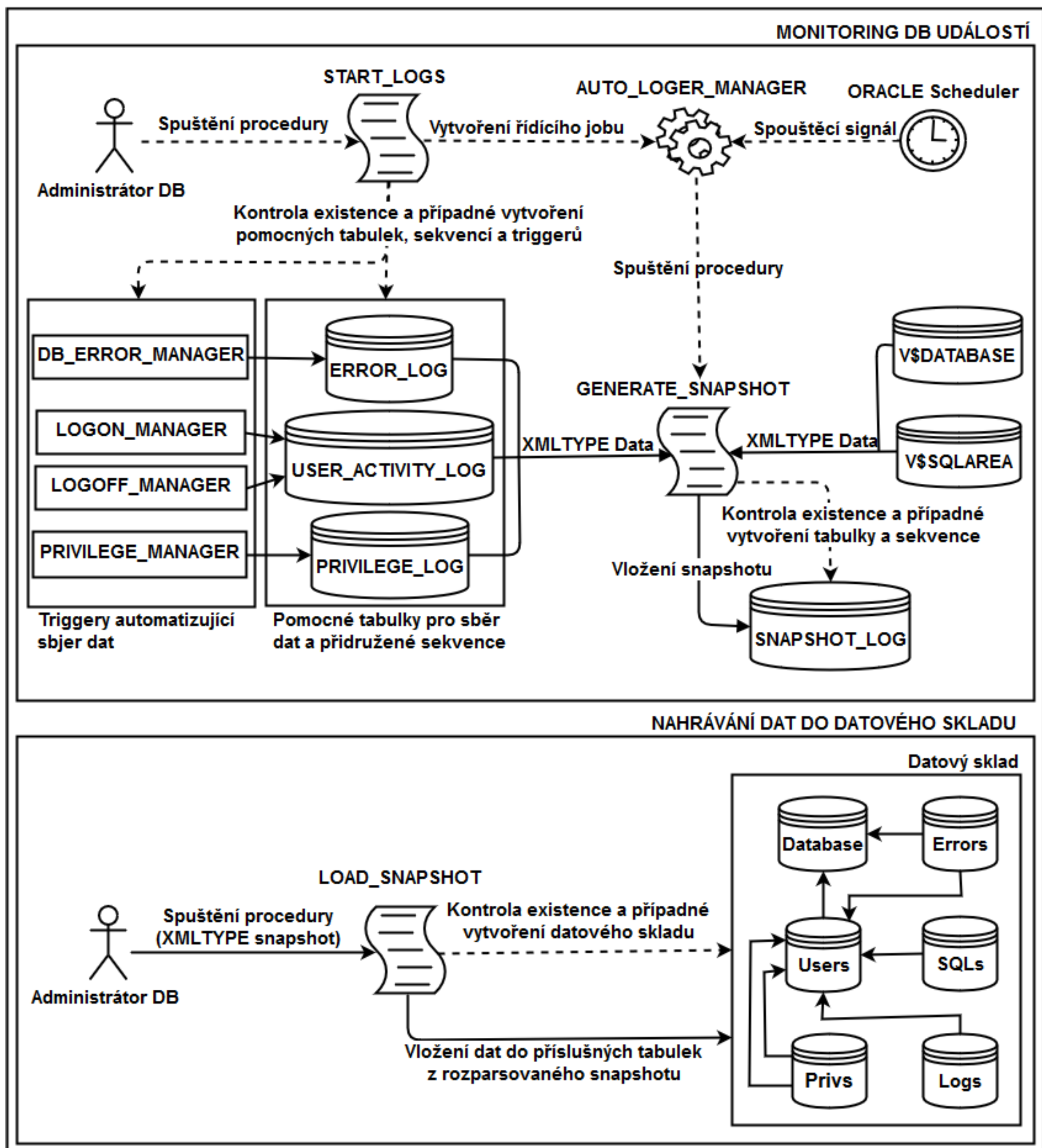
5 Praktická část

Tato část diplomové práce se bude zabývat praktickým využitím teoretických východisek, jež byla popsána v předchozích kapitolách. Popisovaná teoretická východiska budou demonstrována v rámci tvorby modelového balíku, který umožní monitorování různých databázových událostí za pomoci automaticky vytvářených logů. Konkrétní specifikace tohoto balíku bude uvedena dále v části 5.1. Jako prostředí pro tvorbu balíku bude sloužit databázový systém Oracle Database 12c. Funkčnost modelového balíku, a tedy i rozsah veškerých pomocných objektů, je v rámci rozsahu diplomové práce zredukován pouze na funkční ukázku vystihující podstatu popsaných teoretických východisek. Reálný nástroj by pak mohl být rozšířen o další moduly či širší rozsah monitorovaných dat, díky čemuž by bylo možné získat větší povědomí o konkrétní databázi v rámci analytického zkoumání nahromaděných dat. Veškeré zdrojové kódy, jenž budou vytvářeny v rámci praktické části diplomové práce, jsou optimalizovány pro použití v prostředí Oracle Database 12c. Použité zdrojové kódy jsou však zpětně kompatibilní i pro starší verze 11g a 10g, kompatibilita se staršími verzemi již nebyla v rámci diplomové práce testována, nicméně pro starší verze by bylo nutné nahradit použitý systémový balík DBMS_SCHEDULER za již nepoužívaný DBMS_JOBS. Pro vytváření návrhů datových struktur budou vyžity programy Altova XMLSpy a Sybase PowerDesigner 15.

5.1 Popis navrhovaného řešení

V praktické části diplomové práce bude popsán postup při tvorbě modelového databázového balíku pro databázový systém Oracle. Navrhovaný prototyp v sobě bude odrážet potřebu reálných společností po kontrole bezpečnosti, tou se musí zabývat především podniky pracujících s osobními údaji o klientech, jako například banky. Funkčnost vytvářeného balíku bude spočívat v automatizaci monitorování vybraných databázových událostí. Konkrétně půjde o monitoring uživatelských přístupů k databázi, pohyb oprávnění mezi uživateli, dále pak o sledování chyb vznikajících při práci s databází, a nakonec také monitorování spouštěných SQL příkazů. Sbíraná data by bylo možné využít jako jeden z podkladů bezpečnostního auditu zaměřeného a databáze nebo při řešení bezpečnostního incidentu. Z dat vzniklých v rámci monitorování pak budou v pravidelných intervalech automaticky vytvářeny snapshoty ve formátu XML, díky čemuž bude možné tato data snadno využívat nejen v samotných databázích Oracle, ale i na jiných databázových strojích nebo v rámci aplikací napsaných v různých programovacích jazycích či dokonce prostřednictvím webového rozhraní. Posledním funkčním celkem celého balíku pak budou procedury zajišťující automatické vytvoření datového skladu pro monitorovaná data a následné parsování snapshotu

a nahrání dat do vytvořené databázové struktury, tato funkčnost je určena především pro databázi, na níž by docházelo k analýze nasbíraných údajů z více klientských databází, jež by zároveň mohla být propojena s webovým prostředím pomocí podpůrné aplikace vytvořené v APEXu. Přesný popis dat, která budou o jednotlivých událostech ukládána, bude popsán v čísti 5.2.1, v části 5.2.2 pak bude zaměřena na datový model, a tedy na způsob, jak budou data ukládána. Popisovaná funkčnost je znázorněna business schématem na obrázku 9.



Obrázek 9: Business schéma hlavních funkčních celků prototypu [zdroj: autor]

5.2 Návrh datových struktur

Prvním krokem, před tím, než je vůbec možné začít s vytvářením samotného balíku, je navržení a popis jednotlivých datových struktur, jenž budou v rámci balíku vytvářeny a používány. V této části bude nejprve popsána XSD struktura snapshotů, jenž bude sloužit jako vzor pro vytvářené snapshoty v XML formátu. Následně budou definovány dva datové modely, první obsahující pomocné tabulky pro klientskou část, jenž budou sloužit k dočasnému uchování dat před vytvořením snapshotu, druhý pak pro datový sklad, v němž by byla data finálně ukládána a zpracována.

5.2.1 XSD snapshotu

XSD neboli XML Schema Definition je jakousi šablonou, podle které jsou následně vytvářeny XML soubory. Pro účely diplomové práce bude použito XSD k definování podoby snapshotů, jenž budou automaticky v pravidelných intervalech generovány a které v sobě budou obsahovat veškerá analytická data pořízená v rámci dané databáze za předem nastavený časový interval.

Struktura každého snapshotu bude pod kořenovým uzlem „Snapshot" složena ze dvou povinných částí, a to konkrétně z hlavičky a z těla. Hlavička snapshotu bude obsahovat informace o databázi jako takové doplněné o datum a čas vytvoření snapshotu „Timestamp". V rámci rozsahu diplomové práce budou tyto informace omezeny pouze na DBID, název databáze a platformu na níž databáze běží. Tělo snapshotu se pak bude dále rozkládat na čtyři povinné dílčí části reprezentující jednotlivé okruhy informací sbíraných z klientské databáze. Konkrétně se bude jednat o monitorování chyb „DatabaseErrors", uživatelských přístupů „UsersActivities", pohybu oprávnění „PrivilegesActivities" a o monitorování spouštěných SQL příkazů „SQLS". Každá z těchto dílčích částí může obsahovat obsahovat *0* až *n* záznamů o dané aktivitě. Informace obsažené v jednotlivých částech jsou následující:

- „DatabaseErrors" - V rámci monitorování databázových chyb budou ukládány v každém záznamu informace o tom, kdy k danému erroru došlo, kopie chybového zásobníku a jméno uživatele, který danou chybu zapříčinil.
- „UsersActivities" - V rámci monitorování uživatelských přístupů bude každý záznam obsahovat informace o jméně uživatele, datu a času jeho připojení a odpojení od databáze a dále informace o tom, odkud došlo k přístupu do databáze.

- „PrivilegesActivities” - V rámci monitorování pohybu oprávnění budou v každém záznamu sbírána data o tom, kdo provedl konkrétní akci a koho se týkala, zda-li šlo o přidělení či o odebrání oprávnění, o jaké oprávnění šlo a jakého objektu se práva týkala, nakonec bude přidáno datum a čas kdy ke konkrétní akci došlo.
- „SQLS” - V rámci monitorování SQL příkazů budou do snapshotů ukládána data o každém spuštěném SQL. Konkrétně půjde o SQLID, text příkazu, z jakého modulu byl příkaz spuštěn, počet provedení konkrétního příkazu, k jakému množství parsování, řazení a čtení z disku muselo dojít, kolik řádek bylo daným SQL vybráno, jak bylo při vykonávání zatíženo CPU a kolik času zabralo zpracování příkazu, nakonec budou uvedeny informace o tom, kdo daný příkaz spustil a kdy k tomu došlo.

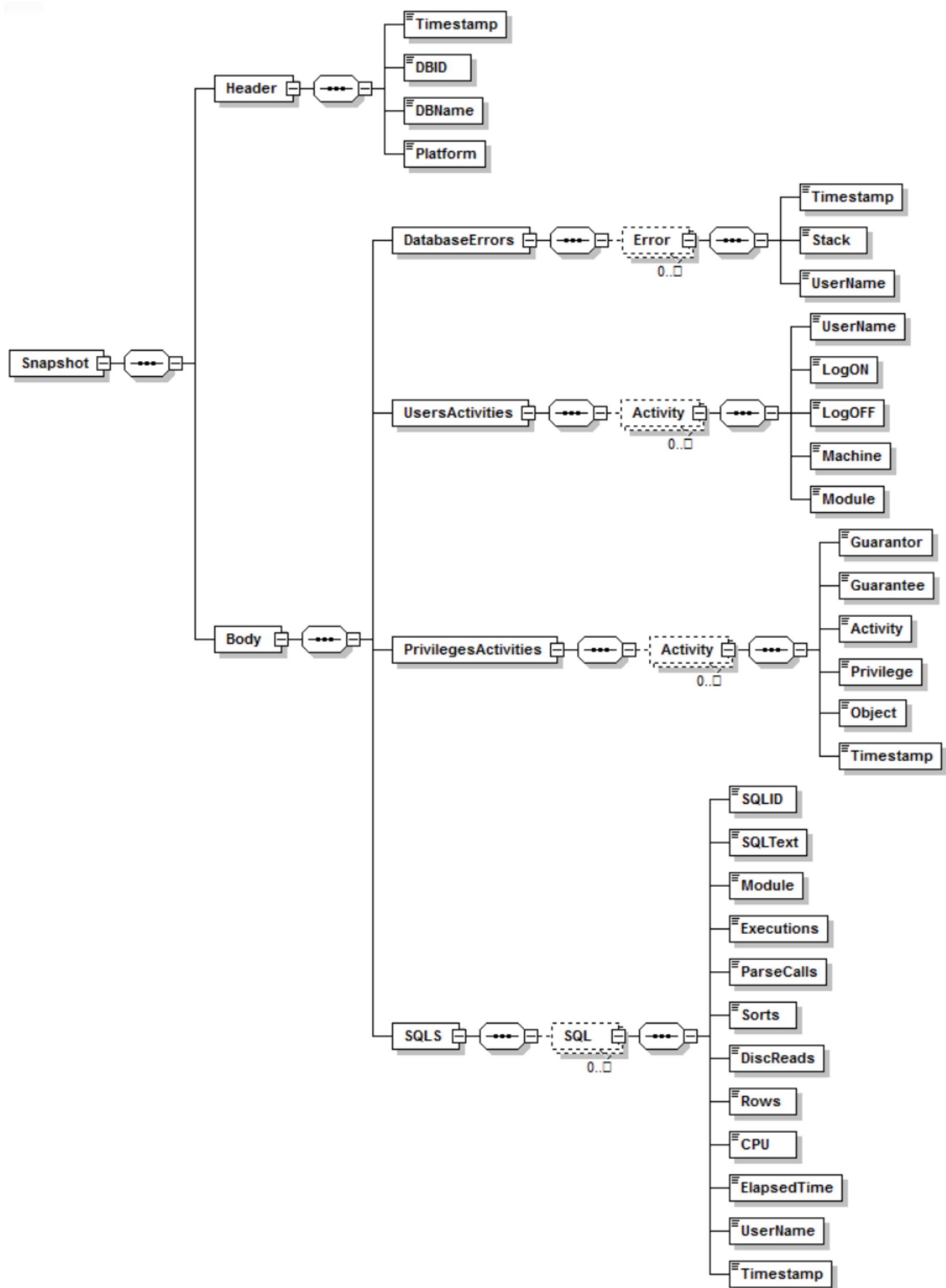
Jak již bylo řečeno na počátku této části, množství dat sbíraných v rámci jednotlivých logů je omezeno vzhledem k rozsahu diplomové práce, v reálném monitorovacím nástroji by pak bylo ukládáno větší množství údajů. Navrhované XSD schéma, jenž bude použito pro tvorbu snapshotů je graficky znázorněno na obrázku 10.

5.2.2 Datový model

Nyní, když byla definována data, s nimiž bude vytvářený balík pracovat, je nezbytné navrhnout tabulky, do kterých budou data ukládána. Jednotlivé tabulky a vazby mezi nimi, stejně tak jako i další podpůrné databázové objekty, pak budou dynamicky vytvářeny či odstraňovány v procedurách a funkcích v rámci vytvářeného balíku, to vše se bude dít automaticky po spuštění monitorovacího nástroje. Tabulky, které budou v této části navrženy, je možné rozdělit do dvou skupin.

5.2.2.1 Pomocné tabulky

První skupinou tabulek, jenž je nutné popsat, jsou pomocné tabulky, do nichž budou data ukládána pouze dočasně před vytvořením snapshotu na klientské databázi. Pro účely vytvářeného balíku budou potřeba pouze čtyři tabulky. První tři tabulky pro budou sloužit k ukládání jednotlivých logů, konkrétně USER_ACTIVITY_LOG pro ukládání dat o uživatelských přístupech k databázi, ERROR_LOG pro ukládání dat o chybách vzniklých v databázi a PRIVILEGE_LOG pro ukládání dat o pohybu oprávnění. Poslední pomocnou tabulkou pro klientskou databázi pak bude SNAPSHOT_LOG, do této tabulky budou ukládány finální snapshoty, odtud pak mohou být exportovány za pomoci systémového balíku



Obrázek 10: XSD schéma snapshotu [zdroj: autor]

DBMS_XMLPROCESSOR do souboru nebo nahrány do datového skladu. Tabulku pro dočasné ukládání logů o spouštěných SQL příkazech není nutné vytvářet, jelikož tyto příkazy již jsou v systému evidovány v požadovaném tvaru a je možné k nim přistupovat za pomoci pohledu v$sqlarea. Jelikož pomocné tabulky budou sloužit pouze k ukládání dat pro konstrukci snapshotů a poté budou data ihned mazána, není nutné mezi nimi vytvářet vazby, jelikož se nepředpokládá, že by nad nimi byly spouštěny nějaké složitější SQL dotazy zahrnující více z nich. Obsah jednotlivých tabulek odpovídá datům definovaným v části 5.2.1, pouze s tím rozdílem, že každému řádku bude navíc přiděleno ID. Graficky je obsah pomocných tabulek demonstrován na obrázku 11.

USER_ACTIVITY_LOG		
ID	number	<pk>
USER_NAME	varchar2(30)	
LOGON	date	
LOGOFF	date	
MACHINE	varchar2(64)	
MODULE	varchar2(64)	

ERROR_LOG		
ID	number	<pk>
TIMESTAMP	date	
USER_NAME	varchar2(30)	
ERROR_STACK	varchar2(4000)	

PRIVILEGE_LOG		
ID	number	<pk>
GUARANTOR	varchar2(30)	
GUARANTEE	varchar2(30)	
ACTIVITY	varchar2(10)	
PRIVILEGE	varchar2(30)	
OBJECT	varchar2(30)	
TIMESTAMP	date	

SNAPSHOT_LOG		
ID	number	<pk>
SNAPSHOT	xmltype	

Obrázek 11: Návrh pomocných tabulek pro měřené DB [zdroj: autor]

5.2.2.2 Datový sklad

Druhá skupina tabulek, s nimiž bude navrhovaný balík pracovat, bude dohromady tvořit datový sklad. Tento datový sklad by byl vytvořen v rámci hlavní databáze, kde by docházelo k analýze veškerých dat získaných na klientských databázích, data budou do datového skladu nahrávána příslušnou procedurou navrhovaného balíku přímo z vytvořených snapshotů. Vzhledem k tomu, že v rámci analýzy dat předpokládáme různé typy dotazů, tak je nutné, aby jednotlivé tabulky v rámci datového skladu byly navzájem logicky provázány za pomoci cizích klíčů a vazeb typu *1:n*, na rozdíl od pomocných tabulek na klientských databázích. V rámci datového skladu budou vytvořeny následující tabulky:

- Database - Tato tabulka v sobě bude obsahovat základní informace o jednotlivých klientských databázích, v rámci kterých bude monitorovací nástroj spuštěn.
- Users - Tato tabulka bude obsahovat jména jednotlivých uživatelů a také jejich příslušnost ke konkrétní databázi, ta bude realizována za pomoci cizího klíče. Jedná se o centrální tabulku, na kterou se budou odkazovat všechny následující tabulky obsahující data o jednotlivých událostech.
- Errors - V rámci této tabulky budou ukládána data o chybách vzniklých v databázi. Vzhledem k očekávaným typům dotazů, v nichž nás ne vždy bude zajímat také uživatel spojený s danou chybou, budou jednotlivé řádky navázány za pomoci cizích klíčů

nejen na uživatele, ale také na konkrétní databázi. Díky tomuto opatření zvýšíme rychlost zpracování dotazů, v nichž se na uživatele nezaměřujeme, a to díky redukci tabulek, které bude třeba spojit v rámci konkrétního dotazu, a tedy i množství dat které bude třeba zapojit.

- Privs - Obsahem této tabulky budou data o pohybu oprávnění, tedy kdo a komu jaká oprávnění přiděloval či odebíral. Jelikož každá aktivita související se změnou oprávnění zahrnuje vždy dvě strany, tedy uživatele, který iniciuje změnu práv a příjemce dané změny, musí i být jednotlivé řádky tabulky „Privs" propojeny s tabulkou „Users" za pomoci dvou cizích klíčů reprezentujících jednotlivé strany pro daný případ.

- Logs - V rámci této tabulky budou ukládána data o uživatelských přístupech k jednotlivým databázím. Obdobně jako u předchozích tabulek, i zde bude nutné vytvořit vazbu na tabulku „Users" za pomoci cizího klíče.

- SQLs - Tato tabulka bude obsahovat data o tom, jaké SQL příkazy byly spouštěny na jednotlivých klientských databázích. I v tomto případě pak je třeba vytvořit vazbu na tabulku „Users" za pomoci cizího klíče.

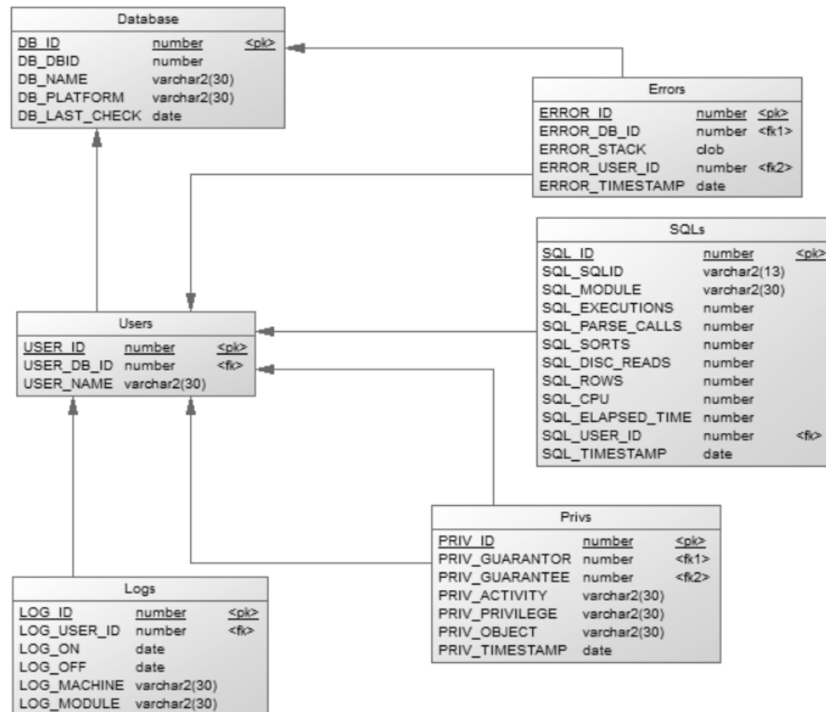
Schéma navrhovaného datového skladu, včetně obsahu jednotlivých tabulek, je graficky znázorněn na obrázku 12.

5.3 Tvorba balíku

Poté, co byly definovány datové struktury, jež budou využívány v rámci balíku zajišťujícího automatické monitorování databázových událostí popsaných v části 5.1, můžeme začít se samotným vytvářením balíku. Vzhledem k rozsahu diplomové práce budou v této části popsány pouze stěžejní celky navrhovaného balíku a některé části s obdobnou funkčností budou zredukovány do jedné konkrétní ukázky, celý kód navrhovaného balíku je pak umístěn v závěru diplomové práce jako příloha.

5.3.1 Hlavička balíku

Jak již bylo zmíněno v části 4.1.5.3, prvním krokem při tvorbě balíku je vytvoření jeho hlavičky a specifikace globálních proměnných a veřejných funkcí a procedur, jež budou moci uživatelé využívat. V rámci navrhovaného balíku nebudou třeba žádné globální proměnné,



Obrázek 12: Návrh datového skladu [zdroj: autor]

nicméně specifikujeme šest veřejných procedur, ostatní funkce a procedury budou uživatelům nepřístupné a budou spouštěny automaticky v případě potřeby. Nejdůležitější z veřejných procedur jsou START_LOGS, jenž bude spouštět celý mechanizmus automatického vytváření logů na klientské databázi a její protipól END_LOGS, která ho naopak zastaví, dále pak procedura LOAD_SNAPSHOT, díky které dojde k nahrání dat ze snapshotu do datového skladu. Zbylé procedury jsou spíše podpůrného charakteru, GENERATE_SNAPSHOT provádí samotné vytvoření snapshotu a je nutné ji definovat jako veřejnou, aby ji mohl spouštět job, jenž bude řídit chod automatického monitorování. Procedury CREATE_DATASTORE a REMOVE_DATASTORE jsou pak zodpovědné za práci s datovým skladem, jsou nastaveny jako veřejné pro případ, že by je uživatel potřeboval využít manuálně. Pro vytvoření hlavičky balíku bude použit následující kód.

```
CREATE OR REPLACE PACKAGE AUTO_LOGER AS
  PROCEDURE GENERATE_SNAPSHOT;
  PROCEDURE START_LOGS (ad_start_date DATE, an_repeat NUMBER DEFAULT 24);
  PROCEDURE END_LOGS (ab_remove_snap BOOLEAN DEFAULT FALSE);
  PROCEDURE CREATE_DATASTORE;
  PROCEDURE REMOVE_DATASTORE;
  PROCEDURE LOAD_SNAPSHOT (ax_snapshot XMLTYPE);
END AUTO_LOGER;
```

5.3.2 Pomocné procedury a funkce

V rámci tvorby těla balíku je nejprve nutné popsat dvě pomocné procedury a jednu pomocnou funkci, ty pak budou využívány v rámci ostatních procedur a funkcí pro práci s databázovými objekty. Tyto pomocné konstrukce není nezbytně nutné vytvářet samostatně, nicméně jejich existence umožní dosáhnout úspory kódu, která by byla progresivně větší s přibývajícím počtem modulů pro monitorování dalších databázových událostí, navíc vzhledem k odstranění opakujících se částí kódu dojde ke zjednodušení případných úprav v kódu.

První z pomocných konstrukcí pro práci s objekty, která zde bude rozebrána, je funkce find_object. Její úlohou bude zjistit, zdali specifikovaný objekt existuje v rámci databáze či nikoli, jelikož v rámci databáze platí pravidlo o jednoznačnosti jmen, bude výsledkem této funkce vždy hodota *0* nebo *1*. Funkce bude mít dva parametry, a to konkrétně jméno objektu a jeho typ, přičemž mezi podporovanými typy objektů budou pouze ty, jenž jsou používány v rámci balíku, tedy triggery, tabulky, sekvence nebo joby. V rámci funkce dojde nejprve k ověření, o jaký typ objektu se jedná, podle toho bude následně upraven předpřipravený příkaz, jenž je uložen v proměnné, k tomu budou sloužit dva příkazy REPLACE, kterými vložíme na příslušná místa nejprve zdrojovou tabulku či pohled a následně konkrétní sloupec. Poté dojde k poslední úpravě příkazu vložením jména objektu, který hledáme. Nakonec dojde ke spuštění předkonstruovaného SQL příkazu za pomoci EXECUTE IMMEDIATE a navrácení výsledné hodnoty. Následující zdrojový kód bude použit v rámci těla balíku pro reprezentaci funkce find_object.

```
FUNCTION find_object (av_object_name VARCHAR2, av_object_type VARCHAR2)
RETURN NUMBER AS
  lv_sql VARCHAR2(1000) := '
    SELECT COUNT(*) FROM #TABLE# WHERE #COLUMN# = ''#NAME#''';
  ln_ret NUMBER;
BEGIN
  IF (av_object_type = 'TRIGGER') THEN
    lv_sql := REPLACE(lv_sql, '#TABLE#', 'dba_triggers');
    lv_sql := REPLACE(lv_sql, '#COLUMN#', 'trigger_name');
  ELSIF (av_object_type = 'TABLE') THEN
    lv_sql := REPLACE(lv_sql, '#TABLE#', 'all_tables');
    lv_sql := REPLACE(lv_sql, '#COLUMN#', 'table_name');
  ELSIF (av_object_type = 'SEQUENCE') THEN
    lv_sql := REPLACE(lv_sql, '#TABLE#', 'all_sequences');
    lv_sql := REPLACE(lv_sql, '#COLUMN#', 'sequence_name');
  ELSIF (av_object_type = 'JOB') THEN
    lv_sql := REPLACE(lv_sql, '#TABLE#', 'all_scheduler_jobs');
    lv_sql := REPLACE(lv_sql, '#COLUMN#', 'job_name');
```

```
  ELSE
    RAISE_APPLICATION_ERROR (-20002,'Wrong object type,
    this procedure supports only types TABLE, TRIGGER, SEQUENCE and JOB');
  END IF;

  lv_sql := REPLACE(lv_sql, '#NAME#', av_object_name);
  EXECUTE IMMEDIATE lv_sql INTO ln_ret;
  RETURN ln_ret;
END;
```

Jako další zde bude popsána procedura create_sequence, úkolem této procedury je vytvářet nové sekvence, které v rámci monitorovacího nástroje budou sloužit pro automatické generování hodnot pro primární klíče. Procedura bude mít jediný parametr, a to jméno sekvence. Způsob, jakým tato procedura bude plnit svůj úkol je následující, vždy nejprve dojde k otestování, zdali již sekvence daného jména existuje, pokud ne, dojde k nahrazení jména v předpřipraveném vzoru příkazu pro tvorbu sekvence a jeho následné spuštění v příkazu EXECUTE IMMEDIATE. Použití tohoto příkazu je nezbytné, jelikož DDL příkazy nelze jiným způsobem v rámci PL/SQL kódu spouštět, toto pravidlo bude velmi důležité i dále v procedurách, jež budou dynamicky vytvářet triggery sbírající data. Obdobně by mohla být vytvořena funkce také pro tvorbu triggerů a tabulek, jež budou v balíku vytvářeny, nicméně zde by úspora kódu nebyla příliš výrazná vzhledem k tomu, že na rozdíl od sekvencí bychom nemohli standardizovat celý CREATE příkaz. Následující zdrojový kód bude použit v rámci těla balíku pro reprezentaci procedury create_sequence.

```
PROCEDURE create_sequence (av_seq_name VARCHAR2) AS
  lv_create VARCHAR2(1000) := '
    CREATE SEQUENCE #NAME#
    MINVALUE 1 MAXVALUE 999999999 INCREMENT BY 1 START WITH 1
    NOCACHE NOORDER NOCYCLE';
BEGIN
  IF (find_object(av_seq_name, 'SEQUENCE') = 0) THEN
    lv_create := REPLACE(lv_create, '#NAME#', av_seq_name);
    EXECUTE IMMEDIATE lv_create;
  END IF;
END;
```

Poslední pomocnou procedurou, jenž zde bude definována je procedura drop_object, ta bude využívána v procedurách END_LOGS a REMOV_DATASTORE, jejím účelem bude sjednotit kód používaný pro odstraňování objektů typu trigger, table nebo sequence. Tato procedura je opět pouze podpůrného charakteru a není nezbytná pro samotný běh balíku, avšak opět zde dojde k úspoře kódu a odstranění opakujících se částí kódu. Procedura drop_object bude mít

dva parametry, a to jméno a typ objektu, který chceme odstranit. Průběh této procedury je následující, nejprve dojde k ověření existence daného objektu, za pomoci funkce find_object definované výše a v případě, že daný objekt je přítomen v databázi, dojde k úpravě příkazu pro smazání daného objektu dvěma příkazy REPLACE. Před konstruovaný příkaz bude následně spuštěn za pomoci EXECUTE IMMEDIATE, čímž dojde k odstranění objektu z databáze. Procedura drop_object je reprezentována následujícím PL/SQL kódem.

```
PROCEDURE drop_object(av_object_name VARCHAR2, av_object_type VARCHAR2)
AS
  lv_sql VARCHAR2(100) := 'DROP #TYPE# #NAME#';
BEGIN
  IF (find_object(av_object_name, av_object_type) = 1) THEN
    lv_sql := REPLACE(lv_sql, '#TYPE#', av_object_type);
    lv_sql := REPLACE(lv_sql, '#NAME#', av_object_name);
    EXECUTE IMMEDIATE lv_sql;
  END IF;
END;
```

5.3.3 Automatizace sběru dat

Jednou z nejdůležitějších částí vytvářeného nástroje určeného k monitorování databázových událostí je jednoznačně sběr dat. K tomuto účelu budou sloužit automatizované spouště, tedy triggery, jenž budou za předem nastavených podmínek ukládat data do pomocných tabulek, ty byly blíže popsány v části 5.2.2.1. Jelikož jsou veškeré vytvářené procedury řešící problematiku automatizace sběru dat založeny na stejném principu, bude v rámci této části popsána jako celek pouze procedura start_privileges_check. Ze zbylých dvou procedur budou následně popsány pouze části související se samotnou automatizací procesu sběru dat, tedy triggery, jenž budou v těchto funkcích vytvářeny.

Procedura start_privileges_check má za úkol připravit veškeré objekty, jež jsou nezbytné pro automatizace sběru dat o pohybu oprávnění v rámci databáze. Prvním krokem bude kontrola, zdali již existuje příslušná pomocná tabulka, do které budou získaná data ukládána, pakliže neexistuje, pak dojde k jejímu vytvoření. Zde je však třeba zmínit, že pokud je třeba v rámci PL/SQL procedury nebo funkce použít DDL příkaz, je nezbytné použít dynamické namísto statického SQL, toho docílíme za pomoci použití příkazu EXECUTE IMMEDIATE. Dalším krokem po vytvoření tabulky je spuštění pomocné procedury create_sequence, čímž dojde k vytvoření sekvence, která bude sloužit ke generování hodnot primárních klíčů pro ukládaná data. Posledním krokem, jenž bude v rámci procedury start_privileges_check proveden je vytvoření triggeru, který zajistí samotnou automatizaci sběru dat.

V případě procedury start_privileges_check se bude jednat o trigger typu AFTER GRANT OR REVOKE ON DATABASE a k jeho spuštění tedy dojde vždy poté, co budou v rámci databáze přidělena či odebrána oprávnění. V rámci tohoto triggeru bude nezbytné využívat některé funkce spjaté se systémovými událostmi. Trigger samotný bude mít specifikovaných několik lokálních proměnných. Dvě z proměnných budou typu ora_name_list_t, jedná se o specifický typ z balíku dbms_standard, jenž umožňuje uchovávat list jmen, konkrétně má podobu tabulky obsahující VARCHAR2(64), proměnné tohoto typu v sobě budou uchovávat názvy oprávnění a uživatelů kterým byly přiděleny či odebrány. Dále budou nutné proměnné uchovávající počty oprávnění a uživatelů jimž byly přiděleny či odebrány. Poslední proměnná bude pak uchovávat jméno uživatele, jenž změnu práv inicializoval.

Po spuštění triggeru pak dojde k následujícím činnostem. Jako první bude použita systémová funkce ora_privilege_list, jenž vrátí počet oprávnění, která byla přidělena či odebrána daným příkazem, jenž spustil trigger, navíc pomocí OUT parametru vrátí také seznam oprávnění, která byla takto použita. Dalším krokem bude rozhodnutí, o jakou systémovou událost šlo, podle toho pak použijeme buď funkci ora_grantee nebo ora_revokee, obě mají podobný efekt jako předchozí zmíněná systémová funkce, jen namísto oprávnění samotných pracuje se jmény uživatelů, kterým byla přidělena v případě ora_grantee nebo odebrána v případě ora_revokee. Poté dojde k získání jména uživatele, jenž inicializoval změnu práv. Posledním krokem pak bude samotné vložení jednotlivých řádků dat do pomocné tabulky vytvořené dříve. Jelikož bylo v daném příkazu použito i oprávnění a j uživatelů, je nutné jednotlivé řádky vkládat za pomoci dvou FOR cyklů omezených počtem oprávnění a uživatelů, jenž byly v předchozích krocích uloženy do proměnných. Tyto dva FOR cykly pak postupně projdou veškeré kombinace oprávnění a uživatelských jmen spjatých s konkrétním příkazem a v každé iteraci dojde k vložení jedné z nich do pomocné tabulky. Následující PL/SQL kód demonstruje, jak bude reprezentována procedura start_privileges_check v rámci vytvářeného balíku.

```
PROCEDURE start_privileges_check AS
BEGIN
  --kontrola existence tabulky a její případné vytvoření
  IF (find_object('PRIVILEGE_LOG', 'TABLE') = 0) THEN
    EXECUTE IMMEDIATE
      'CREATE TABLE PRIVILEGE_LOG (
        ID NUMBER PRIMARY KEY,
        GUARANTOR VARCHAR2(30) NOT NULL,
        GUARANTEE VARCHAR2(30) NOT NULL,
        ACTIVITY VARCHAR2(10) NOT NULL,
        PRIVILEGE VARCHAR2(30) NOT NULL,
        OBJECT VARCHAR2(30) NOT NULL,
```

```
        TIMESTAMP DATE NOT NULL)';
  END IF;

  --kontrola existence sekvence a její případné vytvoření
  create_sequence('PRIVILEGE_LOG_SEQ');

  --vytvoření triggeru
  EXECUTE IMMEDIATE
    'CREATE OR REPLACE TRIGGER PRIVILEGE_MANAGER
      AFTER GRANT OR REVOKE ON DATABASE
    DECLARE
      privileges dbms_standard.ora_name_list_t;
      guarantees dbms_standard.ora_name_list_t;
      ln_privileges_count NUMBER;
      ln_guarantees_count NUMBER;
      lv_guarantor VARCHAR2(30);
    BEGIN
      --získání listu oprávnění v systémovém eventu a jejich počet
      ln_privileges_count := ora_privilege_list(privileges);

      --získání listu uživatelů v systémovém eventu a jejich počet
      IF (ora_sysevent = ''GRANT'') THEN
        ln_guarantees_count := ora_grantee(guarantees);
      ELSE
        ln_guarantees_count := ora_revokee(guarantees);
      END IF;
      --získání jména inicializujícího uživatele
      SELECT user INTO lv_guarantor FROM dual;

      --vložení dat
      FOR i IN 1..ln_privileges_count
      LOOP
        FOR j IN 1..ln_guarantees_count
        LOOP
          INSERT INTO PRIVILEGE_LOG (
              ID, GUARANTOR, GUARANTEE, ACTIVITY,
              PRIVILEGE, OBJECT, TIMESTAMP)
            VALUES (
              PRIVILEGE_LOG_SEQ.NEXTVAL, lv_guarantor,
              guarantees(j), ora_sysevent, privileges(i),
              ora_dict_obj_name, sysdate);
        END LOOP;
      END LOOP;
    END;';
END;
```

Stejný princip jako v proceduře start_privileges_check, tedy ve zkratce vytvoření pomocné tabulky, sekvence a triggeru, je použit i u zbylých procedur zajišťujících automatizaci sběru dat. Proto budou nadále namísto celých procedur popisovány již pouze samotné triggery, které sběr dat fyzicky řídí a jenž jsou nezbytné pro automatizaci těchto procesů, celý kód jednotlivých procedur je pak obsažen v příloze.

V rámci procedury start_errors_check bude vytvářen trigger, k jehož spuštění dojde poté, co nastane v databázi nějaká chyba, z tohoto důvodu se tedy bude jednat o trigger typu AFTER SERVERERROR ON DATABASE. Jediným úkolem tohoto triggeru bude ukládání informací o vzniklé chybě do příslušné pomocné tabulky. Data se vždy budou sestávat z hodnoty primárního klíče, automaticky vygenerované příslušnou sekvencí, aktuální datum a čas získaný z funkce sysdate, jméno uživatele, u kterého došlo ke vzniku chyby a poslední součástí pak bude kopie vrcholu chybového zásobníku. Jméno uživatele získáme ze systémové události sys.login_user. Kopii vrcholu chybového zásobníku pak získáme za pomoci funkce FORMAT_ERROR_STACK, která je součástí balíku DBMS_UTILITY. Následující kód reprezentuje příkaz, jenž bude v rámci procedury vytvářet popisovaný trigger.

```
EXECUTE IMMEDIATE
   'CREATE OR REPLACE TRIGGER DB_ERROR_MANAGER
     AFTER SERVERERROR ON DATABASE
  DECLARE
     ln_runid NUMBER;
  BEGIN
     INSERT INTO ERROR_LOG (ID, TIMESTAMP, USER_NAME, ERROR_STACK)
        VALUES(ERROR_LOG_SEQ.NEXTVAL, sysdate, sys.login_user,
                    DBMS_UTILITY.FORMAT_ERROR_STACK);
  END;';
```

V rámci procedury start_user_check je nutné vytvořit dva triggery, první z nich bude spouštěn po přihlášení uživatele, ten druhý pak před jeho odhlášením. První z triggerů bude typu AFTER LOGON ON DATABASE a jeho úlohou bude ihned po přihlášení libovolného uživatele získat data o tom, kdo a odkud se přihlašuje a tato data pak uložit do příslušné pomocné tabulky. Data, které bude trigger zaznamenávat budou získána převážně z pohledu v$session, jenž nám poskytne informaci o uživatelově jménu, názvu počítače, z nějž došlo k přihlášení a také o programu, který byl k přihlášení použit. Kromě těchto informací bude použit primární klíč, ten bude opět automaticky vygenerován příslušnou sekvencí a dále pak datum přihlášení, to bude reprezentováno návratovou hodnotnou funkce sysdate. Následuje kód reprezentující příkaz vytvářející první z triggrů, jenž bude použit v rámci procedury start_user_check.

```
EXECUTE IMMEDIATE
  'CREATE OR REPLACE TRIGGER LOGON_MANAGER
    AFTER LOGON ON DATABASE
  DECLARE
    lv_user_name VARCHAR2(30);
    lv_machine   VARCHAR2(64);
    lv_module    VARCHAR2(48);
  BEGIN
    SELECT username, machine, module
      INTO lv_user_name, lv_machine, lv_module
      FROM v$session
      WHERE username = (SELECT user FROM dual);
    INSERT INTO USER_ACTIVITY_LOG
      (ID, USER_NAME, LOGON, MACHINE, MODULE)
      VALUES (USER_ACTIVITY_LOG_SEQ.NEXTVAL, lv_user_name,
              sysdate, lv_machine, lv_module);
  END;';
```

Druhý z triggerů, který bude vytvářen procedurou start_user_check, je typu BEFORE LOGOFF ON DATABASE a jeho úlohou bude automatické doplňování informace o datu odhlášení do již vytvořeného záznamu, jenž vznikl za pomoci triggeru popsaného výše. Ke spuštění tohoto triggeru dojde vždy před odhlášením uživatele z databáze a pomocí příkazu UPDATE upraví hodnotu LOGOFF u příslušného řádku, a to konkrétně z *hodnoty null* na návratovou hodnotu funkce sysdate. Jediný řádek, jenž bude upravován, bude nalezen pomocí kombinace uživatelského jména a splnění předpokladu prázdné hodnoty atributu LOGOFF. Tento předpoklad bude fungovat pouze v případě, že je v databázi povolena vždy jen jedna aktivní relace na uživatele, v opačném případě by bylo nutné upravit tabulku tak, aby obsahovala navíc také atribut SID, tedy ID konkrétní relace v rámci této ukázky však budeme předpokládat, že popsaná podmínka je splněna. Následující kód pak reprezentuje příkaz vytvářející druhý z triggerů, jenž bude použit v rámci procedury start_user_check.

```
EXECUTE IMMEDIATE
  'CREATE OR REPLACE TRIGGER LOGOFF_MANAGER
    BEFORE LOGOFF ON DATABASE
  BEGIN
    UPDATE USER_ACTIVITY_LOG
      SET LOGOFF = sysdate
      WHERE USER_NAME = (SELECT user from dual)
        AND LOGOFF IS NULL;
  END;';
```

V rámci automatizovaného monitorovacího nástroje, který je v této diplomové práci popisován, budou dále sbírána také data o spouštěných SQL příkazech. Data o spouštěných SQL příkazech nicméně již v rámci Oracle databázích jsou ukládána a do výsledného snapshotu tedy budou získávána přímo ze systémového pohledu v$sqlarea. Z tohoto důvodu již není třeba vytvářet pro sběr dat o spouštěných SQL příkazech specifickou funkci, která by navíc v databázi pouze hromadila redundantní data. Obdobně by fungovaly veškeré moduly sbírající data ukládaná do systémových tabulek.

5.3.4 Tvorba snapshotů

Nyní, když byl definován postup sběru dat na sledované databázi, je třeba se zaměřit na to, jak z těchto dat vytvořit snapshot ve formátu XML, jenž byl popsán v části 5.2.1 za pomoci XSD schématu. Proces tvorby snapshotů je možné rozdělit na dvě části, a právě těmi se bude zabývat tato část diplomové práce.

První část procesu tvorby snapshotů obsahuje čtyři privátní funkce, jejichž úlohou je konstrukce jednotlivých částí těla definovaného XML snapshotu. Jedná se konkrétně o funkce get_errors, get_privileges, get_sql a get_user, tyto funkce není nezbytně nutné vytvářet samostatně, jelikož jsou použity pouze na jediném místě, a tak nedojde k žádné úspoře kódu, avšak vzhledem k zachování přehlednosti kódu je výhodnější ponechat je oddělené od hlavní procedury vytvářející snapshoty. Zmíněné funkce mají obdobný charakter, všechny jsou volány bez parametrů, jejich návratovým typem je XMLTYPE, jenž bude obsahovat konkrétní část těla snapshotu a taktéž pracují na stejném principu, pouze s dvěma výjimkami, jenž budou zmíněny dále.

Princip, na kterém jsou založeny popisované funkce lze rozdělit do tří kroků. Nejprve dojde k samonému vytvoření XML elementu, jenž reprezentuje konkrétní část těla snapshotu, v závislosti na tom, o jakou funkci se jedná, tento XML element bude následně uložen do proměnné. Tento krok proběhne za pomoci příkazu SELECT, v němž specifikujeme konkrétní XMLELEMENT, ten má tři parametry, z nichž pouze první je povinný, jedná se konkrétně o jméno elementu, atributy a případné další subelementy. V tomto případě nebudou zapotřebí žádné atributy, subelementy pak budou reprezentovány funkcí XMLAGG, jenž umožňuje agregovat fragmenty XML do jednoho celku. Selekce z funkce XMLAGG pak bude použita v klauzuli FROM původního příkazu jako zdroj dat. V rámci subelementů vždy vytvoříme skupinu elementů, jenž budou reprezentovat jednotlivé řádky dat z příslušní tabulky, tyto elementy pak budou mít další subelementy, jenž budou představovat jednotlivé atributy příslušné tabulky. Nejnižší vrstva subelementů pak bude vytvářena za pomoci pří-

kazu XMLFOREST, jenž umožňuje ze zadaných parametrů vytvořit XML fragmenty. Takto vytvořené struktura odpovídá navrhovanému XSD schématu, viz část 5.2.1.

Celý první krok musí být obalen příkazem EXECUTE IMMEDIATE. Nejedná se sice o DDL příkaz, jak jsme popisovali v předchozí části a za normálních okolností by tedy mohl být použit samostatně, avšak každý příkaz pracující s dynamicky vytvořenými tabulkami by znemožnil kompilaci celého balíku, protože v době kompilace balíku ještě dané objekty v databázi neexistují. Příkazem EXECUTE IMMEDIATE si můžeme dovolit obejít tento bezpečnostní mechanismus, jelikož víme, že před spuštěním dané části kódu dojde k automatickému vytvoření příslušných tabulek. Problém může nastat pouze v případě, že by někdo příslušné tabulky odstranil a poté by došlo ke spuštění jedné z těchto procedur, v takovém případě by došlo k chybě ORA-06564 (object does not exist). Vyjímku zde tvoří funkce get_sql, jenž nepoužívá jako zdroj dynamicky vytvořenou tabulku, ale pohled v$sqlarea ve spojení s tabulkou dba_users, proto jako jediná nemusí použít EXECUTE IMMEDIATE.

Druhým krokem v rámci tvorby částí těla snapshotu je pak smazání použitých dat. Tento krok je nezbytný, jelikož není žádoucí, aby se v dalších snapshotech opakovala již transformovaná data. Jak již bylo zmíněno výše, jelikož tyto funkce pracují s dynamicky vytvořenými tabulkami, tak i tento krok musí být obalen pomocí EXECUTE IMMEDIATE. Jedinou vyjímku zde tvoří opět funkce get_sql, v rámci které druhý krok nebude proveden, aby však nedocházelo k opakovanému ukládání dat, budou vždy stažena pouze data pořízena v den, kdy je vytvářen snapshot.

Posledním krokem je samotné ukončení funkce a návrat vytvořeného XML pomocí proměnné, do níž byl uložen. Následující kód znázorňuje funkci get_errors, jenž realizuje výše popsané principy, ostatní funkce vytvářející části těla snapshotu jsou k nalezení v příloze.

```
FUNCTION get_errors RETURN XMLTYPE AS
  lx_return XMLTYPE;
BEGIN
  --vytvoření XML
  EXECUTE IMMEDIATE
    'SELECT XMLELEMENT ("DatabaseErrors", XMLAGG(xml_ret))
      FROM(SELECT XMLAGG(XMLELEMENT("Error",
                XMLFOREST(
                  to_char(timestamp,
                          ''YYYY-MM-DD HH24:MI:SS'') as "Timestamp",
                  error_stack as "Stack",
                  user_name as "UserName")))xml_ret
              FROM error_log)'
    INTO lx_return;
```

```
  --odstranění zabalených dat
  EXECUTE IMMEDIATE 'DELETE FROM error_log';
  RETURN lx_return;
END;
```

Druhou částí, jenž zde bude popisována, je pak procedura GENERATE_SNAPSHOT. Úlohou této procedury je samotné sestavení snapshotu a uložení do pomocné tabulky, toho je docíleno v několika krocích. Prvním krokem je vytvoření jednotlivých částí snapshotu. Nejprve dojde k sestavení hlavičky a její uložení do připravené proměnné, tento postup je obdobný jako u funkcí popsaných výše, s rozdílem, že XML struktura bude vyžadovat pouze jednu úroveň subelementů, a tedy funkce XMLFOREST, bude použita přímo jako parametr funkce XMLELEMENT. Celý příkaz SELECT, obdobně jako u procedury get_sql, nemusí být obalen příkazem EXECUTE IMMEDIATE, jelikož pracujeme se systémovým pohledem v$database a ne s dynamicky vytvořenou tabulkou. Po dokončení hlavičky budou postupně volány funkce popisované výše, a tak dojde k sestavení jednotlivých částí těla a jejich uložení do konkrétních proměnných. Poté, co dojde k sestavení jednotlivých částí musí procedura z těchto částí zkompletovat celý snapshot, to opět proběhne v příkazu XMLELEMENT, jenž bude mít kromě jména dva další parametry, konkrétně půjde o proměnnou obsahující hlavičku snapshotu a dále jednotlivé proměnné obsahující části těla snapshotu uzavřené do dalšího XMLELEMENTu. Posledním krokem je pak ověření, zdali existuje příslušná tabulka a sekvence a jejich případné vytvoření, pak již procedura může nahrát vytvořený snapshot. Následující PL/SQL kód znázorňuje popisovanou proceduru GENERATE_SNAPSHOT.

```
PROCEDURE GENERATE_SNAPSHOT AS
  lx_header XMLTYPE;
  lx_db_errors XMLTYPE;
  lx_privileges XMLTYPE;
  lx_user_activity XMLTYPE;
  lx_sql XMLTYPE;
  lx_snapshot XMLTYPE;
BEGIN
  --vytvoření hlavičky
  SELECT XMLELEMENT ("Header", XMLFOREST (
                to_char(sysdate, 'YYYY-MM-DD HH24:MI:SS') as "Timestamp",
                dbid as "DBID",
                name as "DBName",
                platform_name as "Platform"))
    INTO lx_header
    FROM v$database;
```

```
  --vytvoření těla
  lx_db_errors := get_errors;
  lx_privileges := get_privileges;
  lx_user_activity := get_user;
  lx_sql := get_sql;

  --completace XML
  SELECT XMLELEMENT("Snapshot", lx_header, XMLELEMENT("Body",
                       lx_db_errors,
                       lx_privileges,
                       lx_user_activity,
                       lx_sql))
    INTO lx_snapshot
    FROM dual;

  --kontrola existence tabulky SNAPSHOTS a její případné vytvoření
  IF (find_object('SNAPSHOT_LOG', 'TABLE') = 0) THEN
    EXECUTE IMMEDIATE
      'CREATE TABLE SNAPSHOT_LOG
        (ID NUMBER PRIMARY KEY, SNAPSHOT XMLTYPE NOT NULL)';
  END IF;

  --kontrola existence sekvence a její případné vytvoření
  create_sequence('SNAPSHOT_LOG_SEQ');

  --vložení nového snapshotu
  EXECUTE IMMEDIATE
    'INSERT INTO SNAPSHOT_LOG VALUES (SNAPSHOT_LOG_SEQ.NEXTVAL,:xml)'
    USING lx_snapshot;
  COMMIT;
END;
```

5.3.5 Automatizace monitorovacího nástroje

V předchozích částech byly popisovány jednotlivé procedury umožňující automatizovat konkrétní procesy v rámci vytvářeného monitorovacího nástroje jako například vytváření a mazání objektů, sběr dat a tvorba snapshotů. Nicméně až doposud by bylo nezbytné, aby jednotlivé procedury byly spouštěny ručně a některé, jako například procedura generující snapshoty navíc v pravidelných intervalech. Takovýto proces by byl značně neefektivní, navíc by vyžadoval přítomnost zaměstnance zodpovědného za chod databáze, a to v nestandardních pracovních hodinách, jelikož obdobné procedury sbírající data se zejména na velkých a značně vytížených databázích spouštějí převážně v noci, kdy je zatížení přece jen menší.

V této části bude popsána procedura, jenž umožní automatizovat celý proces monitorování od vytváření objektů, přes sběr dat až po vytváření snapshotů.

Procedura START_LOGS je nejdůležitější částí celého balíku, jejím úkolem je propojit veškeré dosud popsané části dohromady a vytvořit z nich jediný funkční celek. Díky této proceduře navíc dojde ke zjednodušení spouštění a údržby celého procesu, jelikož nyní již bude nutné spustit pouze tuto proceduru a ta veškeré náležitosti automaticky připraví sama.

Procedura START_LOGS bude mít dva parametry. První z parametrů „ad_start_date" definuje, kdy má dojít k prvnímu spuštění procedury generující snapshoty. Druhým parametrem procedury je pak „an_repeat", jenž definuje interval opakování, tedy po jak dlouhé době bude docházet k opětovnému vytvoření dalšího snapshotu, tento parametr je defaultně nastaven na dvacet čtyři. Parametr „an_repeat" je pro jednodušší použití specifikován v hodinách, nicméně drobnou úpravou procedury lze frekvenci měnit.

V rámci procedury START_LOGS jsou nejprve spouštěny jednotlivé dílčí procedury vytvářející pomocné tabulky a s nimi spjaté triggery, jenž mají za úkol sběr dat při konkrétních databázových událostech, tyto procedury byly popsány v části 5.3.3. Dalším krokem je pak ověření, jestli již job řídící vytváření snapshotů v databázi existuje, pokud ano, předpokládáme, že uživatel si přeje upravit tento job s novými parametry, a proto dojde k smazání existujícího jobu. Posledním krokem této procedury je pak vytvoření nového řídícího jobu, tento job ponese název AUTO_LOGER_MANAGER a bude typu STORED_PROCEDURE. Tento typ jobu nám umožní spustit proceduru uloženou v databázi, v rámci vytvářeného jobu se bude jednat konkrétně o proceduru GENERATE_SNAPSHOT. Vytvářenému jobu budou následně přiřazeny ještě další parametry. První parametr se týká se první aktivace jobu, k té dojde v čase určeném dobou počátku (ad_start_date), k níž přičteme interval opakování (an_repeat), ten musíme navíc vydělit hodnotou dvacet čtyři, abychom v rámci formátu DATE dostali hodnotu v hodinách. Druhý parametr se pak týká intervalu opakování. Nakonec dojde ještě k samotnému povolení jobu. Popisovaná procedura bude v balíku reprezentována následujícím kódem.

```
PROCEDURE START_LOGS (ad_start_date DATE, an_repeat NUMBER DEFAULT 24) AS
BEGIN
  --spuštění monitoringu
  start_user_check;
  start_privileges_check;
  start_errors_check;
```

```
  --kontrola existence jobu AUTO_LOGER_MANAGER a jeho případné odstranění
  SELECT COUNT(*) INTO ln_object_check
    FROM all_scheduler_jobs
    WHERE job_name = 'AUTO_LOGER_MANAGER';
  IF (find_object('AUTO_LOGER_MANAGER', 'JOB') = 1) THEN
    DBMS_SCHEDULER.DROP_JOB('AUTO_LOGER_MANAGER');
  END IF;

  --vytvoření nového jobu AUTO_LOGER_MANAGER s novými parametry
  DBMS_SCHEDULER.CREATE_JOB(
    JOB_NAME => 'AUTO_LOGER_MANAGER',
    JOB_TYPE => 'STORED_PROCEDURE',
    JOB_ACTION => 'AUTO_LOGER.GENERATE_SNAPSHOT',
    START_DATE => ad_start_date + (an_repeat / 24),
    REPEAT_INTERVAL => 'FREQ=HOURLY; INTERVAL=' || to_char(an_repeat),
    ENABLED => TRUE);
END;
```

Protipólem procedury START_LOGS je pak procedura END_LOGS, její úlohou je, jak název napovídá, automatizace ukončení veškerých procesů souvisejících monitorováním databázových událostí, to samozřejmě zahrnuje odstranění veškerých dříve vytvořených objektů v takovém pořadí, aby v rámci databáze nedošlo k nějaké chybě. K odstranění objektů bude v případě jobu použita procedura DROP_JOB, která je součástí balíku DBMS_SCHEDULER, pro ostatní objekty se pak bude volat procedura drop_object, jenž byla popsána dříve v části 5.3.2. Procedura END_LOGS bude mít jediný parametr typu BOOLEAN, který určí, zdali má dojít také k odstranění již vytvořených snapshotů, defaultně bude mít parametr hodnotu FALSE a ke smazání tedy nedojde. Procedura samotná odstraní nejprve job dohlížející na pravidelné vytváření snapshotů, poté jednotlivé triggery, tabulky a sekvence, nakonec odstraní také objekty spjaté se snapshoty, ale pouze v případě, že byla procedura volána s hodnotou parametru TRUE. Následující kód zobrazuje popisovanou proceduru.

```
PROCEDURE END_LOGS (ab_remove_snap BOOLEAN DEFAULT FALSE) AS
BEGIN
  --odstranění jobu
  IF (find_object('AUTO_LOGER_MANAGER', 'JOB') = 1) THEN
    DBMS_SCHEDULER.DROP_JOB('AUTO_LOGER_MANAGER');
  END IF;

  --odstranění triggerů
  drop_object('DB_ERROR_MANAGER','TRIGGER');
  drop_object('LOGOFF_MANAGER','TRIGGER');
  drop_object('LOGON_MANAGER','TRIGGER');
```

```
  drop_object('PRIVILEGE_MANAGER','TRIGGER');
  drop_object('SHUTDOWN_MANAGER','TRIGGER');
  drop_object('STARTUP_MANAGER','TRIGGER');

  --odstranění tabulek
  drop_object('ERROR_LOG','TABLE');
  drop_object('DB_ACTIVITY_LOG','TABLE');
  drop_object('PRIVILEGE_LOG','TABLE');
  drop_object('USER_ACTIVITY_LOG','TABLE');

  --odstranění sekvencí
  drop_object('ERROR_LOG_SEQ','SEQUENCE');
  drop_object('DB_ACTIVITY_LOG_SEQ','SEQUENCE');
  drop_object('PRIVILEGE_LOG_SEQ','SEQUENCE');
  drop_object('USER_ACTIVITY_LOG_SEQ','SEQUENCE');

  --odstranění volitelných složek
  IF (ab_remove_snap = TRUE) THEN
    --odstranění tabulek
    drop_object('SNAPSHOT_LOG','TABLE');
    --odstranění sekvencí
    drop_object('SNAPSHOT_LOG_SEQ','SEQUENCE');
  END IF;
END;
```

5.3.6 Práce s datovým skladem

Posledním funkčním celkem, jenž byl definován v části 5.1, je automatizace parsování snapshotů a následné nahrávání získaných dat do předem připraveného datového skladu. Tento funkční celek sestává z několika samostatných procedur a jedné pomocné funkce.

První dvě procedury pracující s datovým skladem jsou určeny k automatizaci jeho konstrukce a odstranění. První z nich je tedy procedura CREATE_DATASTORE tato procedura, jak již název napovídá, má za úkol vytvořit v databázi datový sklad, do kterého budou následně ukládána získaná data. Tato procedura bude fungovat na obdobném principu jako v případě procedur popsaných v části 5.3.3, tedy zkontroluje zdali již objekty existují a pokud ne, pak dojde k jejich vytvoření. Na rozdíl od nich však v rámci této procedury bude vytvářeno více sekvencí a navzájem propojených tabulek, proto je nezbytné zajistit správné pořadí spouštění jednotlivých příkazů EXECUTE IMMEDIATE, dalším rozdílem pak bude absence triggeru. Druhou ze zmíněných procedur je samozřejmě REMOVE_DATASTORE, tato procedura sloužící k automatickému odstranění datového skladu a veškerých objektů s ním spjatých, se

skládá z navzájem logicky navazujících spouštění procedury drop_object, obdobně jako je tomu u procedury END_LOGS popsané v předchozí části diplomové práce. Jelikož jsou obě procedury založeny na již popsaných principech, nebude v této části, vzhledem k rozsahu diplomové práce, uveden jejich PL/SQL kód, nicméně je možné jej najít v příloze.

Jako další zde bude popsána pomocná funkce, jenž bude využívána v rámci procedury parsující jednotlivé snapshoty a nahrávající data do datového skladu. Jedná se konkrétně o funkci get_user_id, tuto funkci není nezbytné vytvářet samostatně, avšak obdobně jako u pomocných procedur popsaných v části 5.3.2, jejím použitím dojde ke značné úspoře kódu a samozřejmě k redukci duplicit v kódu, a tedy zjednodušení budoucích úprav kódu.

Úkolem funkce get_user_id je nalezení hodnoty primárního klíče pro konkrétního uživatele, ten je definován za pomoci dvou vstupních parametrů funkce. Prvním parametrem je hodnota primárního klíče databáze v rámci, které je uživatel zařazen. Tento parametr je nezbytný, přestože v rámci databáze je vždy zaručena unikátnost jmen, v datovém skladu tomu tak být nemusí, jelikož v něm mohou být uložena data z různých databází a nelze tedy zaručit unikátnost uživatelských jmen. Z tohoto důvodu je nezbytné spojit uživatele s konkrétní databází, ve které již pravidlo unikátnosti jmen platí, tím docílíme toho, že nedojde k promíchávání dat z různých databází. Druhým parametrem funkce get_user_id je pak samotné jméno uživatele. Funkce samotná má návratový typ NUMBER, což je v souladu s typem primárního klíče v tabulce USERS, jenž bude vytvořena v rámci datového skladu.

Průběh funkce get_user_id začíná dotazem na počet uživatelů, jenž mají dané jméno a zároveň existují v rámci konkrétní databáze, tento dotaz může nabývat pouze hodnot nula nebo jedna, obdobně jako tomu je u ostatních funkcí testující přítomnost nějakého objektu v databázi, protože zde platí stejné pravidlo unikátních jmen. V případě, že uživatel splňující dané specifikace již v datovém skladu existuje, bude za pomoci dalšího dotazu vybráno jeho ID. Pakliže takový uživatel neexistuje, dojde k vygenerování nového ID za pomoci příslušné sekvence a následnému vytvoření nového záznamu o uživateli. Nakonec bude funkce ukončena a návratová hodnota bude odpovídat konkrétnímu ID uživatele, ať již byla získána kteroukoli cestou. Následující PL/SQL kód reprezentuje funkci get_user_id.

```
FUNCTION get_user_id (an_db_id NUMBER, av_name VARCHAR2)
RETURN NUMBER AS
  ln_row_check NUMBER;
  ln_user_id NUMBER;
BEGIN
```

```
  --kontrola existence záznamu o uživateli
  EXECUTE IMMEDIATE
    'SELECT COUNT(*) FROM USERS
      WHERE USER_DB_ID = :dbid
        AND USER_NAME = :name'
    INTO ln_row_check
    USING an_db_id, av_name;

  --pokud neexistuje, vytvoření nového záznamu a zapamatování ID
  IF (ln_row_check = 0) THEN
    EXECUTE IMMEDIATE
      'SELECT USERS_SEQ.NEXTVAL FROM DUAL' INTO ln_user_id;
    EXECUTE IMMEDIATE
      'INSERT INTO USERS VALUES(:id, :dbid, :name)'
      USING ln_user_id,an_db_id,av_name;

  --pokud existuje vybrání jeho ID
  ELSE
    EXECUTE IMMEDIATE
      'SELECT USER_ID FROM USERS
        WHERE USER_DB_ID = :dbid
          AND USER_NAME = :name'
      INTO ln_user_id
      USING an_db_id, av_name;
  END IF;

  RETURN ln_user_id;
END;
```

Poslední součástí funkčního celku popisovaného v této části diplomové práce je pak procedura LOAD_SNAPSHOT. Úlohou této procedury je samotné parsování snapshotů a nahrávání dat do připraveného datového skladu. Procedura má jediný parametr a tím je samozřejmě samotný snapshot, a to ve formátu XMLTYPE. V rámci procedury je nejprve spuštěna další procedura, a to konkrétně CREATE_DATASTORE, ve které dojde k ověření a případnému vytvoření datového skladu. Poté již může procedura začít se samotným parsováním snapshotu.

Parsování začíná načtením dat o sledované databázi z hlavičky snapshotu. K tomu dojde za pomoci jednoduchého příkazu SELECT v kombinaci s příkazem XMLTABLE, jenž umožňuje ze zadaného XMLTYPE vytvořit tabulku, o jednoduchý příkaz půjde, jelikož v hlavičce snapshotu bude vždy obsažen právě jeden řádek, proto mohou být jednotlivé hodnoty vloženy do proměnných. Pro případ, že by nebyla nalezena žádná data nebo naopak příliš mnoho řádků, bude v části pro zpracování chyb vyvolána výjimka upozorňující na chybný formát

snapshotu. Dále dojde k ověření, zdali již testovaná databáze existuje v datovém skladu, pokud ano, dojde k přepsání data poslední proběhlé kontroly na datum vytvoření snapshotu a pokud ne, pak dojde k vytvoření zcela nového záznamu o databázi.

Když jsou zpracovány údaje z hlavičky snapshotu, dojde následně ke zpracování dat i z dalších částí snapshotu, jenž se zabývají jednotlivými monitorovanými událostmi. Jednotlivé části budou obdobně jako v případě hlavičky zpracovány za pomoci příkazu SELECT v kombinaci s příkazem XMLTABLE. Rozdíl však bude v tom, že konkrétní příkaz bude vždy uzavřen jako podmínka ve FOR cyklu, jelikož zde předpokládáme množství dat v rozmezí *0* až *n*, jak je zřejmé ze struktury snapshotu viz část 5.2.1. V každé iteraci konkrétního cyklu pak vždy dojde k zavolání funkce get_user_id, abychom daný záznam dokázali spojit s konkrétním uživatelem a následně bude vytvořen nový záznam odpovídající jedné zaznamenané události uložené v snapshotu. Tento postup bude stejný pro všechny monitorované události, rozdíl bude pouze v podobě příkazu, jenž bude podmínkou ve FOR cyklu a v příkazu INSERT, ty se budou měnit v závislosti na tom, jakou část snapshotu právě zpracováváme. Následující ukázka PL/SQL kódu znázorňuje část procedury LOAD_SNAPSHOT, v této ukázce jsou zachyceny veškeré popisované kroky, avšak parsování těla snapshotu se omezuje pouze na ukázku zpracování uživatelských přístupů k databázi, celý kód procedury je možné nalézt jako součást přílohy diplomové práce.

```
PROCEDURE LOAD_SNAPSHOT (ax_snapshot XMLTYPE) AS
  ln_db_id NUMBER;
  ln_user_id NUMBER;
  ln_user_id2 NUMBER;
  row_check NUMBER;
  ln_dbid NUMBER;
  lv_db_name VARCHAR2(30);
  lv_platform VARCHAR2(101);
  ld_tmp DATE;
BEGIN
  --kontrola existence a případné vytvoření datového skladu
  CREATE_DATASTORE;
  --parsování snapshotu
  --parsování hlavičky
  SELECT dbid, db_name, platform, to_date(tmp, 'YYYY-MM-DD HH24:MI:SS')
    INTO ln_dbid, lv_db_name, lv_platform, ld_tmp
    FROM xmltable ('/Snapshot/Header' PASSING ax_snapshot COLUMNS
            dbid VARCHAR2(30) path './DBID',
            db_name VARCHAR2(30) path './DBName',
            platform VARCHAR2(101) path './Platform',
            tmp CHAR(21) path './Timestamp');
```

```
--kontrola existence DB a případné vytvoření nového záznamu
EXECUTE IMMEDIATE
  'SELECT COUNT(*) FROM DATABASES
    WHERE DB_DBID = :dbid
      AND DB_NAME = :name'
  INTO row_check
  USING ln_dbid, lv_db_name;
IF (row_check = 0) THEN
  EXECUTE IMMEDIATE
    'SELECT DATABASES_SEQ.NEXTVAL FROM DUAL' INTO ln_db_id;
  EXECUTE IMMEDIATE
    'INSERT INTO DATABASES VALUES(:id,:dbid,:name,:platform,:tmp)'
    USING ln_db_id, ln_dbid, lv_db_name, lv_platform, ld_tmp;
ELSE
  EXECUTE IMMEDIATE
    'SELECT DB_ID FROM DATABASES
      WHERE DB_DBID = :dbid
        AND DB_NAME = :name'
    INTO ln_db_id
    USING ln_dbid, lv_db_name;
  EXECUTE IMMEDIATE
    'UPDATE DATABASES SET DB_LAST_CHECK = :tmp
      WHERE DB_DBID = :ln_dbid
        AND DB_NAME = :lv_db_name'
    USING ld_tmp, ln_dbid, lv_db_name;
END IF;
--parsování přístupových logů
FOR i IN (SELECT username,logon,logoff,machine,module
      FROM xmltable ('/Snapshot/Body/UserActivities/Activity'
        PASSING ax_snapshot COLUMNS
          username VARCHAR2(30) path './UserName',
          logon CHAR(21) path './LogON',
          logoff CHAR(21) path './LogOFF',
          machine VARCHAR2(64) path './Machine',
          module VARCHAR2(64) path './Module'))
LOOP
  --získání uživatelova ID
  ln_user_id := get_user_id(ln_db_id,i.username);
  --vložení nového logu
  EXECUTE IMMEDIATE
    'INSERT INTO LOGS
      VALUES(LOGS_SEQ.NEXTVAL,:userid,:logon,:logoff,:machine,:module)'
    USING ln_user_id,to_date(i.logon, 'YYYY-MM-DD HH24:MI:SS'),
  to_date(i.logoff, 'YYYY-MM-DD HH24:MI:SS'), i.machine, i.module;
END LOOP;
```

```
  --část parsování errorů viz příloha B
  --část parsování oprávnění viz příloha B
  --část parsování sql viz příloha B

/*
pokud hlavička neobsahuje veškerá data,
pak nelze považovat snapshot za přípustný
*/
EXCEPTION
  WHEN NO_DATA_FOUND
    OR TOO_MANY_ROWS THEN
    RAISE_APPLICATION_ERROR(-20001, 'Wrong snapshot format');
END;
```

6 Diskuse a doporučení

V důsledku rozvoje moderní společnosti dochází k rychlému nárůstu množství dat a vzhledem k tomu je logické, že mezi hlavní proudy rozvoje firemních databází patří nejen bezpečnost těchto dat, ale také automatizace procesů, jež s těmito daty pracují. Jak je zřejmé z předchozích kapitol, automatizace procesů v databázových systémech je velmi rozsáhlou problematikou, jenž v sobě zahrnuje mnoho různých dílčích disciplín, z nichž ty nejdůležitější byly popsány v rámci diplomové práce. Avšak vzhledem k šíři daného tématu a rozsahu diplomové práce nemohli být bohužel popsány veškeré techniky využívané při automatizaci procesů.

Praktická část diplomové práce byla zaměřena na demonstraci jednoho z možných využití automatizace procesů v podnikovém prostředí. Předkládané řešení se stávalo z návrhu a vyhotovení balíku řešícího problematiku automatizace monitoringu několika zvolených databázových událostí a práci s nasbíranými daty. Konkrétně byla sbírána data o přístupech uživatelů k databázi, pohybu oprávnění mezi uživateli, vzniku chyb na databázi a o spouštěných SQL příkazech, tato data by v reálném podniku mohla sloužit jako jeden z podkladů pro bezpečnostní audit nebo při řešení bezpečnostních incidentů. Navrhovaný balík byl koncipován tak, aby na něm bylo možné demonstrovat několik různých typů automatizace od vytváření funkcionalit, přes dynamické vytváření databázových objektů, až po využití různých typů triggerů a použití jobu jakožto automatických spouští pro volání uložených funkcionalit. Veškeré techniky použité při konstrukci daného řešení byly popsány v teoretické části, kde byly také specifikovány různé možnosti jejich využití.

Funkční prototyp navržený v předchozí kapitole má sloužit pouze jako ukázka popisovaných principů demonstrující možnosti automatizace, aby bylo možné jej prakticky využít jako monitorovací systém ve firemním prostředí, bylo by nejprve nezbytné rozšířit jeho funkčnost. Jako prostor pro další rozvoj lze označit například doplnění většího množství dat sbíraných o jednotlivých událostech, jejich množství bylo vzhledem k rozsahu práce ve funkčním prototypu zredukováno. Dále by samozřejmě mohli být přidávány další moduly pro monitoring více událostí. Pro usnadnění analýz nasbíraných dat by pak bylo možné vytvořit aplikaci pro přehledné zobrazování nasbíraných dat. Jelikož jsou data převáděna do snapshotů ve formátu XML je možné s nimi pracovat prakticky kdekoli, a tak by bylo možné využít například prostředí Oracle Apex, popřípadě nějaký programovací jazyk schopný komunikovat s databází, jako například Java, C# a podobně, nebo by data moha být reprezentována v rámci webového rozhraní. Nakonec by bylo vhodné také přidat SQL plus skript, jenž by umožnil automatickou instalaci balíku, díky čemuž by bylo možné rychleji nasazovat nové verze.

7 Závěr

Diplomová práce byla zaměřena na problematiku automatizace procesů v databázových systémech Oracle a její využití v podnikatelských subjektech. Hlavním cílem závěrečné práce bylo navržení a realizace funkčního prototypu databázového balíku automatizujícího proces monitoringu vybraných databázových událostí. Dílčími cíli závěrečné práce bylo zmapování aktuálního stavu a vymezení relevantnosti zvolené problematiky, dále pak vytvoření literární rešerše obsahující úvod do problematiky. K naplnění vytyčených cílů závěrečné práce byla použita metodika založena na studiu a analýze dostupných informačních zdrojů a existujících řešení v dané oblasti. Stěžejní pro vypracování této závěrečné práce byly především metody a techniky relačně databázové technologie využívané v kontextu s problematikou automatizace procesů v DBMS Oracle v rámci podnikatelských subjektů, a to zejména programovací jazyk PL/SQL, triggery a joby.

Závěrečná práce vysvětluje důležité pojmy z problematiky automatizace procesů v databázových systémech Oracle a možnosti, které tato problematika poskytuje. Dále poukazuje na důvody, proč je důležité se touto problematikou zabývat a jaké výhody mohou plynout z výsledků její realizace. Hlavním přínosem je pak funkční prototyp navržený v rámci praktické části závěrečné práce. Na vytvářeném prototypu byly prakticky demonstrovány postupy využívané při automatizaci procesů v DBMS Oracle a je možné jej chápat jako návod, jak přistupovat k dané problematice. Pokud by navíc byly provedeny navrhované úpravy, mohl by být tento funkční prototyp reálně nasazen v podnikovém prostředí pro monitoring a analýzu firemních databází a jeho výstupy použity například jako jeden z podkladů pro bezpečnostní audit se zaměřením na databázi.

Ze studia a analýzy vybraných informačních zdrojů a následného návrhu a zpracování vlastního řešení zvolené problematiky dále vyplynuly následující důležité poznatky. V důsledku rozvoje moderní společnosti dochází k rychlému nárůstu množství dat ukládaných v podnikových databázích. Vzhledem k tomuto faktu je logické, že mezi hlavní proudy rozvoje v problematice jejich zpracování patří nejen bezpečnost dat, ale také automatizace procesů, jenž s těmito daty pracují. Tento jev je způsoben především tím, že některé z procesů probíhajících nad podnikovými databázemi již není možné zpracovávat manuálně v reálném čase a s přiměřenými náklady vzhledem k velikosti datové základny a k rychlosti, s jakou tato data přibývají. Dalším z důvodů, proč je problematika automatizace procesů dnes tak populární, jsou výhody, jenž přináší její využití. Díky automatizaci lze například dosáhnout zvýšení rychlosti a plynulosti firemních procesů, usnadnění jejich opakování, zvýšení kontroly nad probíhajícími procesy a odstranění chyb způsobených působením lidského faktoru a mnohé další. Ze zmíněných výhod poté mohou pro podnik plynout i další benefity, jako

například možnost dosáhnout časových a finančních úspor, a především pak schopnost společnosti soustředit své síly na hlavní podnikové činnosti.

Při automatizaci procesu je důležité věnovat velkou pozornost již analýze a návrhu, aby výsledný proces byl co nejvíce robustní, udržovatelný a z bezpečnostního hlediska nezávadný. Při realizaci samotné je nezbytné dbát na čistotu kódu a dodržení norem používaných v konkrétním podniku, což usnadní případné hledání chyb a rozšiřování o další funkcionality. Vhodným postupem je pak psát kód co nejjednodušeji a případné optimalizace ponechat až do fáze, kdy je hotov funkční prototyp.

I přes nesporné výhody, jenž automatizace procesů v databázových systémech přináší, je možné se i dnes v podnikovém prostředí setkat s pasivním přístupem k rozvoji v této oblasti a s nevyužitým potenciálem databázových systémů Oracle. Tento problém se týká zejména menších podniků, jejichž zaměření nesouvisí s oblastí IT, avšak příklady je možné najít i u větších korporací. Nejčastějším nedostatkem pak bývá nevyužití triggerů a jobů pro automatizaci úloh, namísto toho jsou jednotlivé funkcionality často spouštěny ručně, což sebou opět přináší riziko lidské chyby při špatném zřetězení úloh či snížení plynulosti procesu. Tento stav se však pomalu mění, a i menší firmy si začínají uvědomovat nezbytnost automatizace a potenciál jaký skrývá.

Relačně databázová technologie, jejíž základy položil E. F. Codd již v roce 1970, je i dnes stále rozvíjena a přináší nové poznatky v problematice zpracování hromadných dat, mimo jiné jsou rozšiřovány také možnosti automatizace procesů. Rozvoj v této oblasti informačních technologií je stále aktuální a bude jistě pokračovat i do budoucna, jelikož relačně databázová technologie je dnes důležitým základem téměř každého většího podnikového systému, v nichž je dnes automatizace procesů prakticky nezbytností. Navíc není příliš pravděpodobné, že by relačně databázová technologie v podnikových systémech byla v brzké době nahrazena.

Literatura

[1] BRYLA, Bob a Kevin LONEY. Mistrovství v Oracle Database 11g. Brno: Computer Press, 2010. ISBN 978-80-251-2189-4.

[2] FEUERSTEIN, Steven a Bill PRIBYL. Oracle PL/SQL Programming, 6th Edition. O'Reilly Media, 2014. ISBN 978-1-4493-2445-2.

[3] KYTE, Thomas. Oracle : návrh a tvorba aplikací. Brno: CP Books, 2005. ISBN 80-251-0569-5.

[4] LACKO, Ľuboslav. Oracle : správa, programování a použití databázového systému. Brno: Computer Press, 2007. ISBN 978-80-251-1490-2.

[5] LONEY, Kevin. Oracle Database, kompletní průvodce. Brno: Computer press, 2010. ISBN 978-80-251-2489-5.

[6] MCDONALD, Connor a Chaim KATZ. Mastering Oracle PL/SQL: Practical Solutions. Apress, 2004. ISBN 978-1-59059-217-5.

[7] MCLAUGHLIN, Michael. Oracle database 11g PL/SQL programming. New York: McGraw-Hill, 2008. ISBN 978-0-07-149445-8.

[8] PROCHÁZKA, David. Oracle : průvodce správou, využitím a programováním nad databázovým systémem. Praha : Grada, 2009. ISBN 978-80-247-2762-2.

[9] URMAN, Scott a Michael MCLAUGHLIN. Oracle - Programování v PL/SQL. Brno: Computer Press, 2010. ISBN 978-80-251-1870-2.

[10] Oracle Database Documentation. DBMS_SHEDULER Oracle Documentation. [online]. 28.7.2015 [cit. 1.10.2016]. Dostupné z: https://docs.oracle.com/cd/B19306_01/appdev.102/b14258/d_sched.htm#CIHHBGGI

[11] Oracle Database Documentation. Jobs Oracle Documentation. [online]. 10.7.2015 [cit. 1.10.2016]. Dostupné z: https://docs.oracle.com/cd/B28359_01/server.111/b28310/schedover004.htm

[12] Oracle Database Documentation. PL/SQL User Guide and Reference. [online]. 22.3.2005 [cit. 1.10.2016]. Dostupné z: https://docs.oracle.com/cd/B13789_01/appdev.101/b10807.pdf

[13] Oracle Database Documentation. Using Jobs. [online]. 10.7.2015 [cit. 1.10.2016]. Dostupné z: https://docs.oracle.com/cd/B28359_01/server.111/b28310/scheduse002.htm

[14] Oracle Database Documentation. Trigger Documentation. [online]. 28.7.2015 [cit. 1.10.2016]. Dostupné z: https://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_7004.htm

[15] Oracle PL/SQL tutorial. [online]. 1.10.2016 [cit. 1.10.2016]. Dostupné z: https://www.tutorialspoint.com/plsql/plsql_pdf_version.htm

Přílohy

A Definice balíku

```
CREATE OR REPLACE PACKAGE AUTO_LOGER AS
  PROCEDURE GENERATE_SNAPSHOT;
  PROCEDURE START_LOGS (ad_start_date DATE, an_repeat NUMBER DEFAULT 24);
  PROCEDURE END_LOGS (ab_remove_snap BOOLEAN DEFAULT FALSE);
  PROCEDURE CREATE_DATASTORE;
  PROCEDURE REMOVE_DATASTORE;
  PROCEDURE LOAD_SNAPSHOT (ax_snapshot XMLTYPE);
END AUTO_LOGER;
/
```

B Definice těla balíku

```
CREATE OR REPLACE PACKAGE BODY AUTO_LOGER AS
  --část popisující pomocné funkce a procedury
  FUNCTION find_object (av_object_name VARCHAR2, av_object_type VARCHAR2)
  RETURN NUMBER AS
    lv_sql VARCHAR2(1000) := '
      SELECT COUNT(*) FROM #TABLE# WHERE #COLUMN# = ''#NAME#''';
    ln_ret NUMBER;
  BEGIN
    IF (av_object_type = 'TRIGGER') THEN
      lv_sql := REPLACE(lv_sql, '#TABLE#', 'dba_triggers');
      lv_sql := REPLACE(lv_sql, '#COLUMN#', 'trigger_name');
    ELSIF (av_object_type = 'TABLE') THEN
      lv_sql := REPLACE(lv_sql, '#TABLE#', 'all_tables');
      lv_sql := REPLACE(lv_sql, '#COLUMN#', 'table_name');
    ELSIF (av_object_type = 'SEQUENCE') THEN
      lv_sql := REPLACE(lv_sql, '#TABLE#', 'all_sequences');
      lv_sql := REPLACE(lv_sql, '#COLUMN#', 'sequence_name');
    ELSIF (av_object_type = 'JOB') THEN
      lv_sql := REPLACE(lv_sql, '#TABLE#', 'all_scheduler_jobs');
      lv_sql := REPLACE(lv_sql, '#COLUMN#', 'job_name');
    ELSE
      RAISE_APPLICATION_ERROR (-20002,'Wrong object type,
      this procedure supports only types TABLE, TRIGGER, SEQUENCE and JOB');
    END IF;

    lv_sql := REPLACE(lv_sql, '#NAME#', av_object_name);
```

```
    EXECUTE IMMEDIATE lv_sql INTO ln_ret;
    RETURN ln_ret;
  END;

  PROCEDURE create_sequence (av_seq_name VARCHAR2) AS
    lv_create VARCHAR2(1000) := '
      CREATE SEQUENCE #NAME#
      MINVALUE 1 MAXVALUE 999999999 INCREMENT BY 1 START WITH 1
      NOCACHE NOORDER NOCYCLE';
  BEGIN
    IF (find_object(av_seq_name, 'SEQUENCE') = 0) THEN
      lv_create := REPLACE(lv_create, '#NAME#', av_seq_name);
      EXECUTE IMMEDIATE lv_create;
    END IF;
  END;

  PROCEDURE drop_object(av_object_name VARCHAR2, av_object_type VARCHAR2)
  AS
    lv_sql VARCHAR2(100) := 'DROP #TYPE# #NAME#';
  BEGIN
    IF (find_object(av_object_name, av_object_type) = 1) THEN
      lv_sql := REPLACE(lv_sql, '#TYPE#', av_object_type);
      lv_sql := REPLACE(lv_sql, '#NAME#', av_object_name);
      EXECUTE IMMEDIATE lv_sql;
    END IF;
  END;

  --část popisující tvorbu snapshotů
  FUNCTION get_errors RETURN XMLTYPE AS
    lx_return XMLTYPE;
  BEGIN
    --vytvoření XML
    EXECUTE IMMEDIATE
      'SELECT XMLELEMENT ("DatabaseErrors", XMLAGG(xml_ret))
        FROM(SELECT XMLAGG(XMLELEMENT("Error",
                  XMLFOREST(
                    to_char(timestamp,
                            ''YYYY-MM-DD HH24:MI:SS'') as "Timestamp",
                    error_stack as "Stack",
                    user_name as "UserName")))xml_ret
                FROM error_log)'
      INTO lx_return;

    --odstranění zabalených dat
    EXECUTE IMMEDIATE 'DELETE FROM error_log';
```

```
  RETURN lx_return;
END;

FUNCTION get_privileges RETURN XMLTYPE AS
  lx_return XMLTYPE;
BEGIN
  --vytvoření XML
  EXECUTE IMMEDIATE
    'SELECT XMLELEMENT ("PrivilegeActivities", XMLAGG(xml_ret))
      FROM (SELECT XMLAGG(XMLELEMENT("Activity",
                XMLFOREST(
                  guarantor as "Guarantor",
                  guarantee as "Guarantee",
                  activity as "Activity",
                  privilege as "Privilege",
                  object as "Object",
                  to_char(timestamp,
                          ''YYYY-MM-DD HH24:MI:SS'') as "Timestamp")
                ))xml_ret
            FROM privilege_log)'
    INTO lx_return;

  --odstranění zabalených dat
  EXECUTE IMMEDIATE 'DELETE FROM privilege_log';
  RETURN lx_return;
END;

FUNCTION get_sql RETURN XMLTYPE AS
  lx_return XMLTYPE;
BEGIN
  --vytvoření XML
  SELECT XMLELEMENT ("SQLS", XMLAGG(xml_ret))
    INTO lx_return
    FROM (SELECT XMLAGG(XMLELEMENT("SQL",
            XMLFOREST(
              sql_id as "SQLID",
              sql_fulltext as "SQLText",
              module as "Module",
              executions as "Executions",
              parse_calls as "ParseCalls",
              sorts as "Sorts",
              disk_reads as "DiscReads",
              rows_processed as "Rows",
              cpu_time as "CPU",
              elapsed_time as "ElapsedTime",
```

```
                username as "UserName",
                to_char(to_date(first_load_time,
                                'YYYY-MM-DD hh24:mi:ss'),
                        'YYYY-MM-DD hh24:mi:ss') as "Timestamp")
              ))xml_ret
          FROM v$sqlarea JOIN dba_users ON user_id = parsing_user_id
          WHERE to_date(first_load_time, 'YYYY-MM-DD hh24:mi:ss') >
                to_char(SYSDATE, 'dd.mm.yyyy')
            AND module IS NOT NULL);
  RETURN lx_return;
END;

FUNCTION get_user RETURN XMLTYPE AS
  lx_return XMLTYPE;
BEGIN
  --vytvoření XML
  EXECUTE IMMEDIATE
    'SELECT XMLELEMENT ("UserActivities", XMLAGG(xml_ret))
      FROM (SELECT XMLAGG(XMLELEMENT("Activity", XMLFOREST(
              user_name as "UserName",
              to_char(logon, ''YYYY-MM-DD HH24:MI:SS'') as "LogON",
              to_char(logoff, ''YYYY-MM-DD HH24:MI:SS'') as "LogOFF",
              machine as "Machine",
              module as "Module")))xml_ret
            FROM user_activity_log
            WHERE logoff IS NOT NULL)'
    INTO lx_return;

  --odstranění zabalených dat
  EXECUTE IMMEDIATE
    'DELETE FROM user_activity_log WHERE logoff IS NOT NULL';
  RETURN lx_return;
END;

PROCEDURE GENERATE_SNAPSHOT AS
  lx_header XMLTYPE;
  lx_db_errors XMLTYPE;
  lx_privileges XMLTYPE;
  lx_user_activity XMLTYPE;
  lx_sql XMLTYPE;
  lx_snapshot XMLTYPE;
BEGIN
  --vytvoření hlavičky
  SELECT XMLELEMENT ("Header", XMLFOREST (
                to_char(sysdate, 'YYYY-MM-DD HH24:MI:SS') as "Timestamp",
```

```
                 dbid as "DBID",
                 name as "DBName",
                 platform_name as "Platform"))
      INTO lx_header
      FROM v$database;

    --vytvoření těla
    lx_db_errors := get_errors;
    lx_privileges := get_privileges;
    lx_user_activity := get_user;
    lx_sql := get_sql;

    --kompletace XML
    SELECT XMLELEMENT("Snapshot", lx_header, XMLELEMENT("Body",
                            lx_db_errors,
                            lx_privileges,
                            lx_user_activity,
                            lx_sql))
      INTO lx_snapshot
      FROM dual;

    --kontrola existence tabulky SNAPSHOTS a její případné vytvoření
    IF (find_object('SNAPSHOT_LOG', 'TABLE') = 0) THEN
      EXECUTE IMMEDIATE
        'CREATE TABLE SNAPSHOT_LOG
          (ID NUMBER PRIMARY KEY, SNAPSHOT XMLTYPE NOT NULL)';
    END IF;

    --kontrola existence sekvence a její případné vytvoření
    create_sequence('SNAPSHOT_LOG_SEQ');

    --vložení nového snapshotu
    EXECUTE IMMEDIATE
      'INSERT INTO SNAPSHOT_LOG VALUES (SNAPSHOT_LOG_SEQ.NEXTVAL,:xml)'
      USING lx_snapshot;
    COMMIT;
  END;

  --část popisující automatizaci sběru dat
  PROCEDURE start_errors_check AS
  BEGIN
    --kontrola existence tabulky a její případné vytvoření
    IF (find_object('ERROR_LOG', 'TABLE') = 0) THEN
      EXECUTE IMMEDIATE
        'CREATE TABLE ERROR_LOG (
```

```
        ID
        NUMBER PRIMARY KEY,
        TIMESTAMP  DATE NOT NULL,
        USER_NAME  VARCHAR2(30) NOT NULL,
        ERROR_STACK VARCHAR2(4000) NOT NULL)';
  END IF;

  --kontrola existence sekvence a její případné vytvoření
  create_sequence('ERROR_LOG_SEQ');

  --vytvoření servererror triggeru
  EXECUTE IMMEDIATE
    'CREATE OR REPLACE TRIGGER DB_ERROR_MANAGER
      AFTER SERVERERROR ON DATABASE
    DECLARE
      ln_runid NUMBER;
    BEGIN
      INSERT INTO ERROR_LOG (ID, TIMESTAMP, USER_NAME, ERROR_STACK)
        VALUES(ERROR_LOG_SEQ.NEXTVAL, sysdate, sys.login_user,
               DBMS_UTILITY.FORMAT_ERROR_STACK);
    END;';
END;

PROCEDURE start_privileges_check AS
BEGIN
  --kontrola existence tabulky a její případné vytvoření
  IF (find_object('PRIVILEGE_LOG', 'TABLE') = 0) THEN
    EXECUTE IMMEDIATE
      'CREATE TABLE PRIVILEGE_LOG (
        ID NUMBER PRIMARY KEY,
        GUARANTOR VARCHAR2(30) NOT NULL,
        GUARANTEE VARCHAR2(30) NOT NULL,
        ACTIVITY VARCHAR2(10) NOT NULL,
        PRIVILEGE VARCHAR2(30) NOT NULL,
        OBJECT VARCHAR2(30) NOT NULL,
        TIMESTAMP DATE NOT NULL)';
  END IF;

  --kontrola existence sekvence a její případné vytvoření
  create_sequence('PRIVILEGE_LOG_SEQ');

  --vytvoření triggeru
  EXECUTE IMMEDIATE
    'CREATE OR REPLACE TRIGGER PRIVILEGE_MANAGER
      AFTER GRANT OR REVOKE ON DATABASE
```

```
    DECLARE
      privileges dbms_standard.ora_name_list_t;
      guarantees dbms_standard.ora_name_list_t;
      ln_privileges_count NUMBER;
      ln_guarantees_count NUMBER;
      lv_guarantor VARCHAR2(30);
    BEGIN
      --získání listu oprávnění v systémovém eventu a jejich počet
      ln_privileges_count := ora_privilege_list(privileges);

      --získání listu uživatelů v systémovém eventu a jejich počet
      IF (ora_sysevent = ''GRANT'') THEN
        ln_guarantees_count := ora_grantee(guarantees);
      ELSE
        ln_guarantees_count := ora_revokee(guarantees);
      END IF;

      --získání jména inicializujícího uživatele
      SELECT user INTO lv_guarantor FROM dual;

      --vložení dat
      FOR i IN 1..ln_privileges_count
      LOOP
        FOR j IN 1..ln_guarantees_count
        LOOP
          INSERT INTO PRIVILEGE_LOG (
              ID, GUARANTOR, GUARANTEE, ACTIVITY,
              PRIVILEGE, OBJECT, TIMESTAMP)
            VALUES (
              PRIVILEGE_LOG_SEQ.NEXTVAL, lv_guarantor,
              guarantees(j), ora_sysevent, privileges(i),
              ora_dict_obj_name, sysdate);
        END LOOP;
      END LOOP;
    END;';
END;

PROCEDURE start_user_check AS
BEGIN
  --kontrola existence tabulky a její případné vytvoření
  IF (find_object('USER_ACTIVITY_LOG', 'TABLE') = 0) THEN
    EXECUTE IMMEDIATE
      'CREATE TABLE USER_ACTIVITY_LOG (
        ID NUMBER PRIMARY KEY,
        USER_NAME  VARCHAR2(30) NOT NULL,
```

```
          LOGON DATE,
          LOGOFF DATE,
          MACHINE VARCHAR2(64) NOT NULL,
          MODULE VARCHAR2(64) NOT NULL)';
    END IF;

    --kontrola existence sekvence a její případné vytvoření
    create_sequence('USER_ACTIVITY_LOG_SEQ');

    --vytvoření logon triggeru
    EXECUTE IMMEDIATE
      'CREATE OR REPLACE TRIGGER LOGON_MANAGER
        AFTER LOGON ON DATABASE
      DECLARE
        lv_user_name VARCHAR2(30);
        lv_machine   VARCHAR2(64);
        lv_module    VARCHAR2(48);
      BEGIN
        SELECT username, machine, module
          INTO lv_user_name, lv_machine, lv_module
          FROM v$session
          WHERE username = (SELECT user FROM dual);
        INSERT INTO USER_ACTIVITY_LOG
          (ID, USER_NAME, LOGON, MACHINE, MODULE)
          VALUES (USER_ACTIVITY_LOG_SEQ.NEXTVAL, lv_user_name,
                  sysdate, lv_machine, lv_module);
      END;';

    --vytvoření logoff triggeru
    EXECUTE IMMEDIATE
      'CREATE OR REPLACE TRIGGER LOGOFF_MANAGER
        BEFORE LOGOFF ON DATABASE
      BEGIN
        UPDATE USER_ACTIVITY_LOG
          SET LOGOFF = sysdate
          WHERE USER_NAME = (SELECT user from dual)
            AND LOGOFF IS NULL;
      END;';
  END;

  --část popisující automatizaci monitoringu
  PROCEDURE START_LOGS (ad_start_date DATE,
                        an_repeat NUMBER DEFAULT 24) AS
  BEGIN
    --spuštění monitoringu
```

```
    start_user_check;
    start_privileges_check;
    start_errors_check;

    --kontrola existence jobu AUTO_LOGER_MANAGER a jeho případné odstranění
    IF (find_object('AUTO_LOGER_MANAGER', 'JOB') = 1) THEN
      DBMS_SCHEDULER.DROP_JOB('AUTO_LOGER_MANAGER');
    END IF;

    --vytvoření nového jobu AUTO_LOGER_MANAGER s novými parametry
    DBMS_SCHEDULER.CREATE_JOB(
      JOB_NAME => 'AUTO_LOGER_MANAGER',
      JOB_TYPE => 'STORED_PROCEDURE',
      JOB_ACTION => 'AUTO_LOGER.GENERATE_SNAPSHOT',
      START_DATE => ad_start_date + (an_repeat / 24),
      REPEAT_INTERVAL => 'FREQ=HOURLY; INTERVAL=' || to_char(an_repeat),
      ENABLED => TRUE);
END;

PROCEDURE END_LOGS (ab_remove_snap BOOLEAN DEFAULT FALSE) AS
BEGIN
    --odstranění jobu
    IF (find_object('AUTO_LOGER_MANAGER', 'JOB') = 1) THEN
      DBMS_SCHEDULER.DROP_JOB('AUTO_LOGER_MANAGER');
    END IF;

    --odstranění triggerů
    drop_object('DB_ERROR_MANAGER','TRIGGER');
    drop_object('LOGOFF_MANAGER','TRIGGER');
    drop_object('LOGON_MANAGER','TRIGGER');
    drop_object('PRIVILEGE_MANAGER','TRIGGER');
    drop_object('SHUTDOWN_MANAGER','TRIGGER');
    drop_object('STARTUP_MANAGER','TRIGGER');

    --odstranění tabulek
    drop_object('ERROR_LOG','TABLE');
    drop_object('DB_ACTIVITY_LOG','TABLE');
    drop_object('PRIVILEGE_LOG','TABLE');
    drop_object('USER_ACTIVITY_LOG','TABLE');

    --odstranění sekvencí
    drop_object('ERROR_LOG_SEQ','SEQUENCE');
    drop_object('DB_ACTIVITY_LOG_SEQ','SEQUENCE');
    drop_object('PRIVILEGE_LOG_SEQ','SEQUENCE');
    drop_object('USER_ACTIVITY_LOG_SEQ','SEQUENCE');
```

```
  --odstranění volitelných složek
  IF (ab_remove_snap = TRUE) THEN
    --odstranění tabulek
    drop_object('SNAPSHOT_LOG','TABLE');
    --odstranění sekvencí
    drop_object('SNAPSHOT_LOG_SEQ','SEQUENCE');
  END IF;
END;

--část popisující práci s datovým skladem a zpracováním snapshotů
PROCEDURE CREATE_DATASTORE AS
BEGIN
  --kontrola existence tabulek a jejich případné vytvoření
  IF (find_object('DATABASES', 'TABLE') = 0) THEN
    EXECUTE IMMEDIATE
      'CREATE TABLE DATABASES (
        DB_ID NUMBER PRIMARY KEY,
        DB_DBID NUMBER NOT NULL,
        DB_NAME VARCHAR2(9) NOT NULL,
        DB_PLATFORM VARCHAR2(101) NOT NULL,
        DB_LAST_CHECK DATE NOT NULL)';
  END IF;

  IF (find_object('USERS', 'TABLE') = 0) THEN
    EXECUTE IMMEDIATE
      'CREATE TABLE USERS (
        USER_ID NUMBER PRIMARY KEY,
        USER_DB_ID NUMBER NOT NULL,
        USER_NAME VARCHAR2(30) NOT NULL,
        FOREIGN KEY (USER_DB_ID) REFERENCES DATABASES(DB_ID))';
  END IF;

  IF (find_object('ERRORS', 'TABLE') = 0) THEN
    EXECUTE IMMEDIATE
      'CREATE TABLE ERRORS (
        ERROR_ID NUMBER PRIMARY KEY,
        ERROR_DB_ID NUMBER NOT NULL,
        ERROR_STACK VARCHAR2(4000) NOT NULL,
        ERROR_USER_ID NUMBER NOT NULL,
        ERROR_TIMESTAMP DATE NOT NULL,
        FOREIGN KEY (ERROR_DB_ID) REFERENCES DATABASES(DB_ID),
        FOREIGN KEY (ERROR_USER_ID) REFERENCES USERS(USER_ID))';
  END IF;
```

```
IF (find_object('SQLS', 'TABLE') = 0) THEN
  EXECUTE IMMEDIATE
    'CREATE TABLE SQLS (
      SQL_ID NUMBER PRIMARY KEY,
      SQL_SQLID VARCHAR2(13) NOT NULL,
      SQL_TEXT CLOB NOT NULL,
      SQL_MODULE VARCHAR2(64) NOT NULL,
      SQL_EXECUTIONS NUMBER NOT NULL,
      SQL_PARSE_CALLS NUMBER NOT NULL,
      SQL_SORTS NUMBER NOT NULL,
      SQL_DISC_READS NUMBER NOT NULL,
      SQL_ROWS NUMBER NOT NULL,
      SQL_CPU NUMBER NOT NULL,
      SQL_ELAPSED_TIME NUMBER NOT NULL,
      SQL_USER_ID NUMBER NOT NULL,
      SQL_TIMESTAMP DATE NOT NULL,
      FOREIGN KEY (SQL_USER_ID) REFERENCES USERS(USER_ID))';
END IF;

IF (find_object('PRIVS', 'TABLE') = 0) THEN
  EXECUTE IMMEDIATE
    'CREATE TABLE PRIVS (
      PRIV_ID NUMBER PRIMARY KEY,
      PRIV_GUARANTOR NUMBER,
      PRIV_GUARANTEE NUMBER,
      PRIV_ACTIVITY VARCHAR2(30),
      PRIV_PRIVILEGE VARCHAR2(30),
      PRIV_OBJECT VARCHAR2(30),
      PRIV_TIMESTAMP DATE,
      FOREIGN KEY (PRIV_GUARANTOR) REFERENCES USERS(USER_ID),
      FOREIGN KEY (PRIV_GUARANTEE) REFERENCES USERS(USER_ID))';
END IF;

IF (find_object('LOGS', 'TABLE') = 0) THEN
  EXECUTE IMMEDIATE
    'CREATE TABLE LOGS (
      LOG_ID NUMBER PRIMARY KEY,
      LOG_USER_ID NUMBER,
      LOG_ON DATE,
      LOG_OFF DATE,
      LOG_MACHINE VARCHAR2(64),
      LOG_MODULE VARCHAR2(64),
      FOREIGN KEY (LOG_USER_ID) REFERENCES USERS(USER_ID))';
END IF;
```

```
    --kontrola existence sekvencí a jejich případné vytvoření
    create_sequence('DATABASES_SEQ');
    create_sequence('USERS_SEQ');
    create_sequence('ERRORS_SEQ');
    create_sequence('SQLS_SEQ');
    create_sequence('PRIVS_SEQ');
    create_sequence('LOGS_SEQ');
  END;

  PROCEDURE REMOVE_DATASTORE AS
  BEGIN
    --odstranění tabulek
    drop_object('LOGS','TABLE');
    drop_object('PRIVS','TABLE');
    drop_object('SQLS','TABLE');
    drop_object('ERRORS','TABLE');
    drop_object('USERS','TABLE');
    drop_object('DATABASES','TABLE');

    --odstranění sekvencí
    drop_object('LOGS_SEQ','SEQUENCE');
    drop_object('PRIVS_SEQ','SEQUENCE');
    drop_object('SQLS_SEQ','SEQUENCE');
    drop_object('ERRORS_SEQ','SEQUENCE');
    drop_object('USERS_SEQ','SEQUENCE');
    drop_object('DATABASE_SEQ','SEQUENCE');
  END;

  FUNCTION get_user_id (an_db_id NUMBER, av_name VARCHAR2)
  RETURN NUMBER AS
    ln_row_check NUMBER;
    ln_user_id NUMBER;
  BEGIN
    --kontrola existence záznamu o uživateli
    EXECUTE IMMEDIATE
      'SELECT COUNT(*) FROM USERS
        WHERE USER_DB_ID = :dbid
          AND USER_NAME = :name'
      INTO ln_row_check
      USING an_db_id, av_name;

        --pokud neexistuje vytvoření nového záznamu a uložení ID
    IF (ln_row_check = 0) THEN
      EXECUTE IMMEDIATE
        'SELECT USERS_SEQ.NEXTVAL FROM DUAL' INTO ln_user_id;
```

```
      EXECUTE IMMEDIATE
        'INSERT INTO USERS VALUES(:id, :dbid, :name)'
        USING ln_user_id,an_db_id,av_name;

  --pokud existuje uložení jeho ID
  ELSE
    EXECUTE IMMEDIATE
      'SELECT USER_ID FROM USERS
        WHERE USER_DB_ID = :dbid
          AND USER_NAME = :name'
      INTO ln_user_id
      USING an_db_id, av_name;
  END IF;
  RETURN ln_user_id;
END;

PROCEDURE LOAD_SNAPSHOT (ax_snapshot XMLTYPE) AS
  ln_db_id NUMBER;
  ln_user_id NUMBER;
  ln_user_id2 NUMBER;
  row_check NUMBER;
  ln_dbid NUMBER;
  lv_db_name VARCHAR2(30);
  lv_platform VARCHAR2(101);
  ld_tmp DATE;
BEGIN
  --kontrola existence datového skladu a jeho případné vytvoření
  CREATE_DATASTORE;

  --parsování snapshotu
  --parsování hlavičky
  SELECT dbid, db_name, platform, to_date(tmp, 'YYYY-MM-DD HH24:MI:SS')
    INTO ln_dbid, lv_db_name, lv_platform, ld_tmp
    FROM xmltable ('/Snapshot/Header' PASSING ax_snapshot COLUMNS
            dbid VARCHAR2(30) path './DBID',
            db_name VARCHAR2(30) path './DBName',
            platform VARCHAR2(101) path './Platform',
            tmp CHAR(21) path './Timestamp');

  --kontrola existence záznamu o DB a jeho případné vytvoření
  EXECUTE IMMEDIATE
    'SELECT COUNT(*) FROM DATABASES
      WHERE DB_DBID = :dbid
        AND DB_NAME = :name'
    INTO row_check
```

```
    USING ln_dbid, lv_db_name;
  IF (row_check = 0) THEN
    EXECUTE IMMEDIATE
      'SELECT DATABASES_SEQ.NEXTVAL FROM DUAL' INTO ln_db_id;
    EXECUTE IMMEDIATE
      'INSERT INTO DATABASES VALUES(:id,:dbid,:name,:platform,:tmp)'
      USING ln_db_id, ln_dbid, lv_db_name, lv_platform, ld_tmp;
  ELSE
    EXECUTE IMMEDIATE
      'SELECT DB_ID FROM DATABASES
        WHERE DB_DBID = :dbid
          AND DB_NAME = :name'
      INTO ln_db_id
      USING ln_dbid, lv_db_name;
    EXECUTE IMMEDIATE
      'UPDATE DATABASES SET DB_LAST_CHECK = :tmp
        WHERE DB_DBID = :ln_dbid
          AND DB_NAME = :lv_db_name'
      USING ld_tmp, ln_dbid, lv_db_name;
  END IF;

  --parsování uživatelských přístupů
  FOR i IN (SELECT username,logon,logoff,machine,module
        FROM xmltable('/Snapshot/Body/UserActivities/Activity'
          PASSING ax_snapshot COLUMNS
            username VARCHAR2(30) path './UserName',
            logon CHAR(21) path './LogON',
            logoff CHAR(21) path './LogOFF',
            machine VARCHAR2(64) path './Machine',
            module VARCHAR2(64) path './Module'))
  LOOP
    --získání uživatelova ID
    ln_user_id := get_user_id(ln_db_id,i.username);

    --vložení nového logu
    EXECUTE IMMEDIATE
      'INSERT INTO LOGS
        VALUES(LOGS_SEQ.NEXTVAL,:userid,:logon,:logoff,:machine,:module)'
      USING ln_user_id,to_date(i.logon, 'YYYY-MM-DD HH24:MI:SS'),
    to_date(i.logoff, 'YYYY-MM-DD HH24:MI:SS'), i.machine, i.module;
  END LOOP;

  --parsování errorů
  FOR i IN (SELECT username,stack,tmp
        FROM xmltable('/Snapshot/Body/DatabaseErrors/Error'
```

```
        PASSING ax_snapshot COLUMNS
          username VARCHAR2(30) path './UserName',
          stack VARCHAR2(4000) path './Stack',
          tmp CHAR(21) path './Timestamp'))
LOOP
  --získání uživatelova ID
  ln_user_id := get_user_id(ln_db_id,i.username);

  --vložení nového záznamu o erroru
  EXECUTE IMMEDIATE
    'INSERT INTO ERRORS
      VALUES(ERRORS_SEQ.NEXTVAL,:db,:stack,:userid,:tmp)'
    USING ln_db_id, i.stack, ln_user_id,
          to_date(i.tmp, 'YYYY-MM-DD HH24:MI:SS');
END LOOP;

--parsování oprávnění
FOR i IN (SELECT guarantor,guarantee,activity,privilege,object,tmp
      FROM xmltable ('/Snapshot/Body/PrivilegeActivities/Activity'
        PASSING ax_snapshot COLUMNS
          guarantor VARCHAR2(30) path './Guarantor',
          guarantee VARCHAR2(30) path './Guarantee',
          activity VARCHAR2(30) path './Activity',
          privilege VARCHAR2(30) path './Privilege',
          object VARCHAR2(30) path './Object',
          tmp CHAR(21) path './Timestamp'))
LOOP
  --získání uživatelova ID
  ln_user_id := get_user_id(ln_db_id,i.guarantor);
  ln_user_id2 := get_user_id(ln_db_id,i.guarantee);

  --vložení nového záznamu o pohybu oprávnění
  EXECUTE IMMEDIATE
    'INSERT INTO PRIVS
      VALUES(
        PRIVS_SEQ.NEXTVAL,:guarantor,:guarantee,
        :activity,:privilege,:obj,:tmp)'
    USING
      ln_user_id,ln_user_id2,i.activity,i.privilege,
      i.object,to_date(i.tmp, 'YYYY-MM-DD HH24:MI:SS');
END LOOP;

--parsování sql
FOR i IN (
  SELECT
```

```
        sqlid ,sql_text ,module,executions ,parse_calls ,sorts ,
        disk_reads ,num_rows,cpu,elapsed_time ,username ,tmp
        FROM xmltable ('/Snapshot/Body/SQLS/SQL'
          PASSING ax_snapshot COLUMNS
            sqlid VARCHAR2(13) path './SQLID',
            sql_text CLOB path './SQLText',
            module VARCHAR2(64) path './Module',
            executions NUMBER path './Executions',
            parse_calls NUMBER path './ParseCalls',
            sorts NUMBER path './Sorts',
            disk_reads NUMBER path './DiscReads',
            num_rows NUMBER path './Rows',
            cpu NUMBER path './CPU',
            elapsed_time NUMBER path './ElapsedTime',
            username VARCHAR2(30) path './UserName',
            tmp CHAR(21) path './Timestamp'))
    LOOP
      --získání uživatelova ID
      ln_user_id := get_user_id(ln_db_id,i.username);

      --vložení nového záznamu o sql
      EXECUTE IMMEDIATE
        'INSERT INTO SQLS
          VALUES(
            SQLS_SEQ.NEXTVAL,:sqlid ,:text ,:module,
            :executions ,:calls ,:sorts ,:reads ,:num_rows,
            :cpu,:elapsed_time , :userid , :tmp)'
        USING
          i.sqlid ,i.sql_text ,i.module,i.executions ,
          i.parse_calls ,i.sorts ,i.disk_reads ,i.num_rows,
          i.cpu, i.elapsed_time ,ln_user_id ,
          to_date(i.tmp,'YYYY-MM-DD HH24:MI:SS');
    END LOOP;
  /*
  pokud hlavička neobsahuje veškerá data,
  pak nelze považovat snapshot za přípustný
  */
  EXCEPTION
    WHEN NO_DATA_FOUND
      OR TOO_MANY_ROWS THEN
      RAISE_APPLICATION_ERROR(-20001, 'Wrong snapshot format');
  END;
END AUTO_LOGER;
/
```