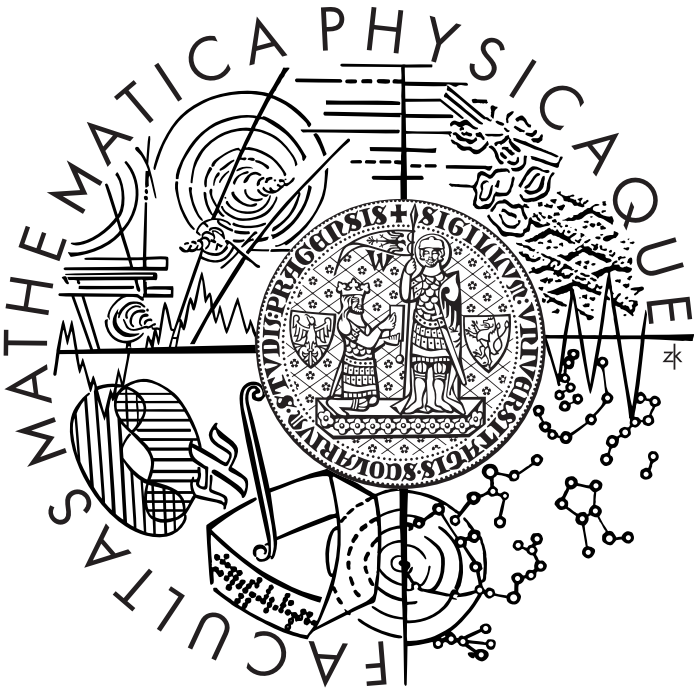
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



BRANISLAV HUDEC

Správa verzí databázových schémat
Database schema version control

Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Michal Kopecký, Ph.D.
Studijní program: Informatika, obor Softwarové systémy

Děkuji RNDr. Michalovi Kopeckému, Ph.D. za cenné rady, náměty a konzultace, kterými přispěl k napsání této diplomové práce. Dále děkuji svým rodičům a celé rodině za dlouhodobou podporu, bez které by tato práce nemohla nikdy vzniknout.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 16.4.2008 Branislav Hudec

Název práce: Správa verzí databázových schémat

Autor: Branislav Hudec

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Michal Kopecký, Ph.D.

E-mail vedoucího: Michal.Kopecky@mff.cuni.cz

Abstrakt: Hlavním cílem práce je navrhnout a implementovat nástroj na zjišťování rozdílů ve dvou databázových schématech. Praktické uplatnění takového nástroje je poměrně široké. Nejčastějším případem použití však bude synchronizace dvou databázových schémat, která původně vznikla stejným vytvořujícím skriptem, tj. na začátku byla schémata shodná. Postupem času uživatel různými operacemi zasahuje do jednoho či druhého schématu, což následně vede k jejich rozdílné struktuře. V jistém okamžiku může uživatel opět požadovat, aby schémata obsahovala stejné databázové objekty, tj. aby byla, co se týče struktury, opět shodná. Nástroj na správu verzí databázových schémat v prvním kroku porovná databázové objekty v obou schématech a zjistí rozdíly. V dalších krocích vygeneruje různé formy výstupu pro uživatele – grafické znázornění změn, XML výstup a textový výstup. Zřejmě nejzajímavější formou výstupu je SQL skript, který po vykonání v jedné z databází zabezpečí, že obě schémata budou synchronizovaná. To znamená, že budou obsahovat stejné databázové objekty a tedy opět dojde ke sjednocení verzí databázových schémat.

Klíčová slova: synchronizace schemat, relační schema, diferenční algoritmus, SQL, XML, Java

Title: Database Schema Version Control

Author: Branislav Hudec

Department: Department of Software Engineering

Supervisor: RNDr. Michal Kopecký, Ph.D.

Supervisor's e-mail address: Michal.Kopecky@mff.cuni.cz

Abstract: The main goal of this diploma thesis is to design and implement an application that would be capable of searching for differences between two database schemas. There are many uses for such tool, but there is one that is the most common. Two database schemas are created with the same SQL script, so they are equal at the beginning. A user can later add or modify database objects in one of the schemas, so the schemas may not be equal anymore. The implemented tool can find the differences between the schemas and present the results to the user in various formats (diff tree in GUI, XML, and text). The most valuable form of output is undoubtedly an SQL script. The purpose of the SQL script is the synchronization of the two schemas. The user can execute such script in a database containing the second schema, and the second schema will be modified according to the first schema.

Keywords: schema synchronisation, relational schema, diff algorithm, SQL, XML, Java

Obsah

1 ÚVOD

Neodmysliteľnou súčasťou každého informačného systému je *perzistenčná vrstva*, ktorá sa stará o ukladanie *aplikačných dát* na pevný disk (prípadne na iné pamäťové médium) a ich opätovné nahranie do operačnej pamäte. Tieto dáta zostávajú uložené na pamäťovom médiu aj po reštarte aplikácie, prípadne celého systému.

Vzhľadom na to, že aplikačné dáta sú vo väčšine prípadov štruktúrované a dajú sa dekomponovať na bežné dátové typy, programátori aplikácií zvyčajne neimplementujú vlastnú perzistenčnú vrstvu, ale spoliehajú sa na existujúce riešenia, ktoré takéto dáta dokážu uložiť a spravovať. Najčastejším prípadom býva použitie tzv. *relačnej databázy*. Existuje veľa implementácií od rôznych dodávateľov, ktoré sa podstatne líšia v množstve ponúkaných funkcií, v spôsobe interného ukladania a spracovania dát, v počte a kvalite obslužných nástrojov, apod. Na druhej strane ale môžeme medzi implementáciami nájsť veľa spoločných čŕt – napríklad podobný užívateľský pohľad na organizáciu dát (rovnaké užívateľské objekty – tabuľky, záznamy v tabuľkách, stĺpce, atď.), rovnaký základ jazyka, ktorý sa používa na manipuláciu s dátami, atď. Výber vhodnej relačnej databázy pre danú aplikáciu býva jedným z najdôležitejších rozhodnutí, ktoré musí riešiteľský tím urobiť. Prechod na inú relačnú databázu väčšinou nie je ani priamočiary, ani triviálny a je lepšie sa mu vyhnúť.

Kvôli prísnemu štrukturovaniu a typovaniu dát v relačných databázach stojí programátor často pred úlohou zmeniť typ dát, ktoré sa do databázy ukladajú – napríklad zistí, že povolená dĺžka textového reťazca nepostačuje a treba ju zväčšiť. Keďže väčšinou existuje niekoľko databáz, s ktorými aplikácia pracuje (testovacia databáza, produkčná databáza, atď.), hrozí nebezpečenstvo, že sa zmeny neprevedú vo všetkých databázach naraz. Tento problém je významný hlavne pri inkrementálnom spôsobe vývoja aplikácie, kde sa štruktúra dát mení často, spolu s pribúdajúcou funkcionalitou systému. Riešiteľské tímy si preto musia spolu s *verziami aplikácie* uchovávať aj *verzie databázových schém* (tj.

verzie štruktúr databázy) a pri aktualizácii verzie programového kódu vykonať aj aktualizáciu databázovej schémy a to vo všetkých databázach, s ktorými aplikácia pracuje.

V predchádzajúcich odsekoch sme načrtli dva problémy, s ktorými sa stretávajú autori informačných systémov. Obidva problémy sa dajú riešiť manuálne. Prvý – menej častý problém (export databázovej schémy v aktuálnej databáze do databázy iného výrobcu) – vyriešime jednorázovým manuálnym prepísaním vytvárajúcich skriptov do iného (často podobného) jazyka a ich spustením na cieľovom databázovom stroji. Druhý problém (zmeny v relačnej schéme počas vývoja a počas prevádzkovania aplikácie) sa vyskytuje častejšie a môžeme ho vyriešiť podrobným dokumentovaním akejkoľvek zmeny, ktorú na niektorej databáze vykonáme. Takéto záznamy sa neskôr môžu analyzovať a pre každú databázu sa môže vytvoriť aktualizačný príkaz (*patch*), ktorý databázové schémy zosynchronizuje.

Tieto postupy sú zvyčajne veľmi náchylné na chyby. Navyše sú relatívne pracné a pri každej (aj minimálnej) zmene ich treba opakovať. Hlavná motivácia tejto práce je teda zrejmá – pokúsiť sa zbaviť autorov informačných systémov tejto manuálnej práce a navrhnúť konfigurovateľný nástroj, ktorý umožní automatické nachádzanie zmien v relačných schémach dvoch databáz a navrhne vhodné kroky, ktoré budú viesť k ekvivalentným schémam. Špeciálnym prípadom bude porovnávanie schémy existujúcej databázy s prázdnou databázou (ktorá môže byť od iného výrobcu). Vykonaním navrhnutých zmien v prázdnej databáze vlastne zreplikujeme existujúcu relačnú schému z iného databázového stroja a použijeme pritom jazyk (dialekt) cieľovej databázy.

1.1 Štruktúra práce

V závere úvodnej kapitoly si objasníme význam pojmov, ktoré budeme v tejto práci často používať. Motivácii, ktorá nás viedla k napísaniu práce, sa budeme venovať v kapitole číslo 2. V tretej kapitole podrobne zanalyzujeme problém, predstavíme aplikácie, ktoré riešia synchronizáciu iným spôsobom a vysvetlíme, prečo sme sa rozhodli vytvoriť vlastný systém na synchronizáciu databázových schém. Štvrtá kapitola predstaví technológie, ktoré budeme v práci používať. Nasledovať bude popis implementácie v jazyku *Java*, ktorej sa venujeme v piatej kapitole. Pomerne dôležitou časťou celého navrhnutého systému je grafický klient, vlastná aplikácia s názvom *DbStudio*. Tento klient ponúka jednoduchý a pohodlný spôsob ovládania celej synchronizácie. K dispozícii sú aj ďalšie funkcie na prácu s databázami. Podrobný návod na používanie *DbStudia* nie je kvôli jeho rozsahu súčasťou práce, čitateľ ho ale môže nájsť na priloženom CD.

1.2 Používané pojmy

V celej práci budeme používať niektoré pojmy, ktoré sa vzťahujú hlavne k databázovým technológiám. V tejto kapitole si ich postupne vysvetlíme.

Databáza (skratka *DB*) je uložená množina štruktúrovaných dát [1]. Skratkou *DBMS* (z angl. *Database management system*) označujeme databázu spolu so súborom programov, ktorými ovládame spôsob uloženia dát v databáze a ktorými dáta z databázy opätovne získavame [2]. Výrobcovia databázových riešení spravidla takéto programy priložia do inštalačných balíkov, preto často hovoríme napr. o *DBMS Oracle*, alebo o *DBMS MySQL* a máme na mysli kompletné databázové riešenia od firiem *Oracle*, resp. *MySQL*. *Dátový model* je spôsob, akým databáza ukladá informácie. Poznáme niekoľko typov dátových modelov, napríklad *objektový dátový model*, *objektovo-relačný dátový model*, atď. My sa budeme zaoberať databázami s *relačným dátovým modelom*, ktorý je založený na matematickej definícii *relácie*. Databázy, ktoré na ukladanie dát používajú relačný model, sa nazývajú *relačné databázy*.

Základným prvkom relačnej databázy je *tabuľka*, ktorá reprezentuje reláciu. Každá *tabuľka* má jednoznačne určený názov a zoznam stĺpcov. Tabuľka slúži ako úložisko štruktúrovaných dát, ktoré do nej vkladáme formou *záznamov* (riadkov). Okrem tabuliek databázy obsahujú množstvo ďalších objektov, napr. *obmedzenia* (constraints), *indexy* (indexes), *pohľady* (views), atď. Súhrnne ich budeme nazývať *databázové objekty*.

Databázová schéma je množina databázových objektov, ktoré sú navzájom logicky prepojené. Schéma predstavuje oddelené a nezávislé úložisko databázových objektov v rámci jednej databázy. Ak sú dva objekty rovnako pomenované, ale nachádzajú sa v dvoch rôznych *schémach*, nekolidujú spolu. V *DBMS Oracle* štandardne patrí schéma užívateľovi s rovnakým menom.

Požiadavku na dáta, ktoré spĺňajú isté podmienky, budeme volať *dotaz* (angl. query). *Dotazovať* sa môže užívateľ prostredníctvom niektorej z priložených aplikácii, ktoré sú súčasťou inštalačného balíka DBMS, alebo prostredníctvom aplikácie tretej strany. Môže tiež použiť rozhranie databázy a dotazovať sa z programového kódu vlastnej aplikácie.

V relačných databázach sa ako dotazovací jazyk spravidla používa niektorý z dialektov jazyka *SQL* (*Structured Query Language*). SQL je štandardizovaný jazyk, viac o existujúcich štandardoch čitateľ nájde v [3]. Prvý všeobecne uznávaný štandard bol *SQL-86*, neskôr ho nahradil štandard *SQL-92* [4]. Napriek tomu, že existujú aj novšie štandardy (napríklad SQL:1999 [5], ktorý do SQL zavádza objektové prvky, alebo najnovší *SQL:2003*

– rozdiely oproti SQL:1999 sú uvedené napríklad v [6]), nie všetky databázy ich podporujú. Pri práci s databázami rôzneho typu budeme teda využívať štandard SQL-92. Pri práci s databázami toho istého typu budeme pripúšťať použitie vlastného dialektu jazyka SQL.

2 MOTIVÁCIA

Hlavná myšlienka tejto práce už bola stručne naznačená v úvode. V tejto sekcii podrobne rozoberieme dôvody, pre ktoré sme sa rozhodli riešiť problém synchronizácie databázových schém.

Softwarové aplikácie väčšinou riešia problém, ktorý vznikol v reálnom svete. Či už ide o program na účtovanie financií, webovskú stránku študijného informačného systému, alebo o rezervačný systém leteniek, software vždy len prináša pohľad na skutočný svet a snaží sa ho interpretovať svojim používateľom. A keďže skutočný svet sa často mení, je prirodzené, že sa spolu s ním musí meniť aj informačný systém. Napríklad po zmene zákona musí účtovnícky software evidovať údaje, ktoré predtým neboli potrebné. Alebo rozdelením magisterského štúdia na bakalárske a nadväzujúce magisterské vznikla v študijnom informačnom systéme potreba evidovať nové údaje o študentoch. Bez ohľadu na kvalitu počiatočnej analýzy softwarového diela sú zmeny počas jeho fungovania nevyhnutné.

Zmeny aplikácie priamo vyvolávajú zmeny v databáze, ktorú aplikácia používa. Takéto zmeny môžu byť jednoduché (premenovanie databázového objektu, pridanie stĺpca do tabuľky), alebo oveľa zložitejšie (rozdelenie tabuľky do dvoch a pridanie vzťahovej tabuľky, modifikácia *view*, atď.). A práve tieto zmeny databázových objektov nás budú zaujímať predovšetkým.

Jedna aplikácia spravidla ukladá svoje dáta do jednej databázovej schémy, preto sa budeme venovať zmenám databázových objektov v rámci jednej, pevne definovanej schémy. V prípade, že aplikácia využíva na ukladanie dát viacero schém, synchronizáciu prevedieme na každej schéme osobitne. Preto budeme často hovoriť aj o *synchronizácii databáz* a budeme tým myslieť synchronizáciu jednej, alebo viacerých schém týchto dvoch databáz. Zaujímať nás budú predovšetkým zmeny v štruktúre databázy, menej sa budeme venovať zmenám samotných dát.

Keďže pri akejkoľvek zmene v aplikácii (bez ohľadu na jej komplexnosť) sa musí nová funkčnosť otestovať, nebýva zvykom, aby sa priamo upravovala reálna, tzv. *produkčná*

databáza. Programátorský tím má väčšinou k dispozícii niekoľko testovacích databáz, ktoré majú rovnakú štruktúru, ako reálna databáza, databázy sa však môžu líšiť obsahom (dátami). Programátor teda upraví kód aplikácie aj štruktúru testovacej databázy, otestuje novú funkčnosť a ak je spokojný s výsledkom, nahradí starú aplikáciu novou verziou a zároveň upraví štruktúru produkčnej databázy rovnakým spôsobom, ako pri vývoji upravoval svoju testovaciu databázu. Tento proces je samozrejme oveľa komplexnejší a náročnejší, väčšinou vyžaduje krátku odstávku systému a preto sa v rámci jednej aktualizácie vykoná niekoľko zmien naraz – nasadí sa nová verzia systému. Keďže zmeny sa môžu týkať rôznych častí systému, môžu na nich paralelne pracovať rôzne programátorské tímy. Môže sa stať, že pre jeden systém existuje niekoľko rôznych testovacích databáz (každý tím má svoju vlastnú) a v istom momente sa všetky svojou štruktúrou líšia od produkčnej databázy, navyše sa odlišujú aj navzájom.

V praxi nastáva aj iný prípad, kedy sa štruktúra produkčnej databázy zmení oproti testovacím databázam. V prípade závažnej chyby v kóde aplikácie, alebo v samotných dátach, dochádza k výpadku funkčnosti celého systému, ktorý treba vo veľmi krátkom čase vyriešiť. Častým prípadom chyby napríklad býva príliš „prísne“ obmedzenie hodnôt, ktoré sa môžu nachádzať v stĺpci tabuľky. Pri vložení hodnoty, s ktorou analýza aplikácie nepočítala, nastane chyba, ktorá sa dá vyriešiť zrušením, resp. zmiernením obmedzenia. Ak sa táto zmena spätne neprenesie aj do testovacích databáz, existuje riziko, že v produkčnej databáze po nasadení novej verzie opäť pribudne pôvodné obmedzenie a opäť sa prejaví pôvodná chyba.

Je určite správnou stratégiou podrobne zaznamenávať všetky zmeny, ktoré programátor uskutoční na testovacej, alebo na produkčnej databáze. Tieto zaznamenané zmeny by sa však mali porovnávať s výsledkom automatizovanej analýzy, ktorá analyzuje testovaciu a produkčnú databázu, resp. dve testovacie databázy medzi sebou. Štruktúra referenčnej databázy sa porovná so štruktúrou porovnávanej databázy a výsledkom je zoznam objektov, ktoré sa nachádzajú v jednej databáze a zároveň sa nenachádzajú v druhej databáze, prípadne sa nachádzajú v obidvoch databázach, ale majú v nich rôzne vlastnosti. Výsledok porovnania môže mať rôzne podoby – grafický výstup pre programátora, alebo databázového administrátora, textový výstup (napríklad kvôli archivácii výstupov automatizovanej analýzy), alebo strojovo spracovateľný XML výstup, ktorý sa dá použiť napríklad pri vytvorení podrobnej dokumentácie zmien (konverziou do HTML, alebo PDF). Asi najzaujímavejší výstup z analýzy by ale bol synchronizačný SQL kód. Po spustení kódu v *porovnávanej databáze* by sa táto databáza dostala do (z hľadiska štruktúry databázových

objektov) rovnakého stavu, ako *referenčná databáza*. SQL kód musí zohľadňovať SQL dialekt *porovnávanej databázy*, ako aj rozdiely medzi rôznymi databázami (napr. v dátových typoch).

2.1 Cieľ práce

Cieľom práce je analyzovať, navrhnúť a implementovať systém, ktorý bude porovnávať dve databázové schémy. Mal by byť zrozumiteľný hlavne pre vývojárov databázových aplikácii a databázových administrátorov, ktorí ovládajú jazyk SQL a jeho dialekty a v menšej miere ovládajú aj nejaký ďalší programovací jazyk. Dôraz sa preto kladie na jednoduchú konfigurovateľnosť systému a jeho flexibilitu (práca s rôznymi dialektmi SQL). Dôležitá je možnosť spúšťať systém automaticky (tj. možnosť spustiť porovnávanie z konzoly, alebo z plánovača úloh, ako je napr. nástroj *cron* v OS Unix).

Vstupné údaje môže systém získať priamymi dotazmi do databázy, mal by ale podporovať aj *offline použitie*, pri ktorom načíta skôr uložené informácie o stave databázových schém k danému okamžiku.

Systém by mal podporovať niekoľko druhov výstupov – výstup určený pre človeka (text) a výstup určený na ďalšie strojové spracovanie (XML). Súčasťou výstupu by mala byť aj sekvencia SQL príkazov, ktorá po vykonaní v jednej z databáz zabezpečí rovnakú štruktúru v obidvoch databázových schémach.

3 ANALÝZA PROBLÉMU

V tejto kapitole detailne popíšeme problém synchronizácie dvoch databázových schém a navrhneme jeho riešenie. V úvode sa budeme zaoberať existujúcimi riešeniami a vysvetlíme, prečo sme sa rozhodli implementovať vlastný systém. Pomenujeme hlavné ciele, podľa ktorých sme postupovali pri návrhu aplikácie a naznačíme, kde vidíme jej primárne použitie. Účelom kapitoly je poskytnúť podrobný náhľad na problematiku, ale príliš nezabiehať do implementačných detailov, ktoré rozoberieme v samostatnej kapitole.

3.1 Existujúce riešenia

Problém synchronizácie databázových schém je rovnako starý, ako samotné databázy. Je preto prirodzené, že existuje niekoľko nástrojov, ktoré tento problém pomáhajú riešiť. Podrobnejšie sa budeme venovať dvom z nich a to nástroju *Schemagic*, ktorý je výsledkom diplomovej práce na MFF UK [7] a komerčnému produktu *TOAD for Oracle* [8] od firmy *Quest*.

3.1.1 Schemagic

Aplikácia *Schemagic* sa dá logicky rozdeliť do dvoch celkov. Prvá časť sa stará o získavanie dát, ktoré popisujú určené schémy, druhá časť dokáže na základe istých predpisov porovnať dve schémy a vygenerovať synchronizačný SQL skript. Popis funkcionality je teda v podstate zhodný s funkcionalitou, ktorú očakávame od našej aplikácie.

Nevýhodou *Schemagicu* ako synchronizačného nástroja je to, že synchronizačný skript sa vytvorí tzv. *XSL transformáciou* (XSLT) dvoch XML dokumentov, popisujúcich referenčnú a porovnávanú schému. XSL transformácia je spôsob, ako skonvertovať strom pôvodného XML dokumentu do stromu cieľového dokumentu na základe istých pravidiel [9]. Keďže XSLT je univerzálna technológia, umožňujúca pokročilé transformácie ľubovoľných XML dokumentov, konfigurácia synchronizácie je pomerne ťažkopádna.

Užívateľ nemôže dostatočne flexibilne reagovať na rôzne vlastnosti objektov, problémom je napríklad presné zadefinovanie, kedy sú dva databázové objekty v dvoch schémach ekvivalentné. Väčšia flexibilita je potrebná aj na prípadnú konverziu dátových typov medzi databázovými strojmi, alebo na určenie pozície vygenerovaných SQL príkazov v rámci výsledného skriptu. Vývojári databázových aplikácii nemusia dostatočne dobre poznať princíp XSL transformácií a všetky možnosti, ktoré im táto technológia ponúka, preto hrozí, že nebudú vedieť nakonfigurovať synchronizáciu tak, aby výsledný skript spĺňal ich predstavy.

V ďalšom texte uvidíme, že napriek tomu, že synchronizačná funkcionalita *Schemagicu* nespĺňa naše požiadavky, časť, ktorá sa stará o získanie metadát je úplne postačujúca a náš systém ju bude využívať.

3.1.2 Toad for Oracle

Toad for Oracle [8] je univerzálny nástroj na prácu s DBMS Oracle. Funkcia, ktorá umožňuje synchronizovať databázové schémy je dostupná len v plnej, platenej verzii. Porovnávať sa dajú len schémy toho istého databázového stroja (DBMS Oracle), čo je istá nevýhoda. Užívateľ môže sám určiť, ktoré objekty sa budú synchronizovať. V ponuke je veľa špecifických objektov pre DBMS Oracle, napr. adresáre (*directories*), triedy jazyka Java, materializované pohľady (*materialized views*), atď. Užívateľ môže využiť aj ďalšie funkcie, napríklad sa môže porovnávať počet riadkov tabuliek, apod. *Toad* ponúka niekoľko druhov výstupov – strom zmenených objektov v grafickom rozhraní, textový popis zmenených objektov, aj synchronizačný SQL skript. Výsledná podoba skriptu však nie je konfigurovateľná, užívateľ nemôže ovplyvniť, ako bude tento skript vyzerať. Práve nedostatočná konfigurovateľnosť a zameranie len na jednu DBMS je dôvodom, prečo sme sa napriek existencii takéhoto nástroju rozhodli vytvoriť vlastný systém.

3.1.3 Ostatné riešenia

Existujú aj ďalšie riešenia, ktorých úlohou je synchronizovať databázové schémy. Spomenieme napríklad *EMS DB Comparer for MySQL* [10] (riešenie pre DBMS MySQL), alebo *DB Solo* [11]. Z nášho pohľadu však nástroje nespĺňajú aspoň niektoré z nasledujúcich požadovaných vlastností:

- multiplatformový nástroj, spustiteľný na bežne rozšírených operačných systémoch
- možnosť porovnávania schém rôznych databázových strojov

- maximálna konfigurovateľnosť nástroja, kedy užívateľ môže podstatne ovplyvniť, ako bude vyzerať výstup
- open-source licencia, prípadne freeware

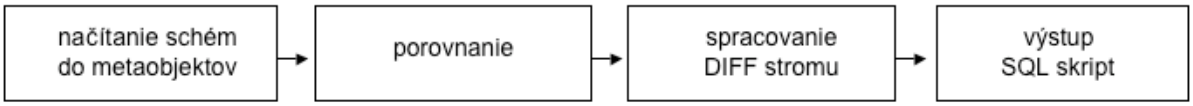
Preto sme sa rozhodli implementovať synchronizáciu z iného pohľadu, ako uvedené nástroje. Ako však neskôr uvidíme, nezačínali sme od začiatku, ale využili sme základ, ktorý poskytuje aplikácia *Schemagic*.

3.2 Základný model funkcionality

Prvým krokom užívateľa je vždy výber dvoch databázových schém, ktoré chce porovnať. Obidve sa môžu nachádzať v jednej databáze, alebo môžu byť umiestnené na rôznych databázových strojoch. Systém dovoľuje porovnávať databázové schémy, ktoré existujú v databázach dvoch rôznych výrobcov.

Prvá vybraná databázová schéma je základná – tzv. *referenčná* (reference). Druhá vybraná databázová schéma bude tzv. *porovnávaná* (comparison). Užívateľ ako referenčnú schému vyberie tú, ktorú považuje za cieľovú. Tj. chce zistiť, ako sa *porovnávaná* databázová schéma líši od *referenčnej*. Ak neskôr spustí v *porovnávanej databáze* sekvenciu SQL príkazov, ktorú získa ako výstup synchronizačnej aplikácie, štruktúra porovnávanej schémy sa bude rovnať štruktúre referenčnej schémy.

Postupnosť operácií, ktoré sú potrebné na porovnanie databázových schém a na získanie synchronizačného SQL skriptu je znázornená na nasledujúcom obrázku:



Obr.: Základné operácie synchronizačnej aplikácie

Porovnávanie databázových objektov bude realizované v jazyku Java, preto bude nutné načítať všetky informácie o obidvoch databázových schémach do pamäte. Na tento účel si vytvoríme vlastné Java triedy, ktoré budú reprezentovať stromovú štruktúru databázovej schémy. Tieto dva stromy budeme potom porovnávať v dvoch fázach – v prvej určíme, ktoré objekty sú ekvivalentné, v druhej porovnáme ich vlastnosti. Nakoniec dva stromy zlúčime do jedného. Takýto zlúčený strom budeme volať *diff strom*. Na základe tohto stromu vygenerujeme rôzne formy výstupu. Postupne si podrobne rozoberieme, čo jednotlivé operácie znamenajú a akým spôsobom ich budeme implementovať v aplikácii.

3.3 Štruktúra databázovej schémy

V prvej kapitole sme zadefinovali databázovú schému ako množinu databázových objektov, ktoré spolu logicky súvisia, tj. jedná sa o úložný priestor pre databázové objekty. Každá aplikácia môže mať v rámci jednej databázy svoju vlastnú databázovú schému a objekty z tejto schémy nebudú žiadnym spôsobom ovplyvňovať databázové objekty z iných schém, ktoré môžu používať iné aplikácie. Pri pripájaní sa k databáze musí užívateľ väčšinou definovať, akú schému chce používať. Ak chce po prihlásení pristupovať k objektom z inej databázovej schémy, musí použiť kvalifikované meno objektu, ktoré obsahuje meno schémy, aj meno samotného objektu. Jeden užívateľ má spravidla prístup k objektom z práve jednej schémy, nemusí to však platiť, ak mu administrátor, alebo vlastník povolí prístup aj k iným objektom.

Databázová schéma má stromovú štruktúru. Koreň stromu tvorí samotná *schéma*, ktorá ďalej obsahuje *tabuľky*, *pohľady*, *procedúry*, *funkcie*, atď. Tabuľka je opäť predkom ďalších uzlov – obsahuje totiž *stĺpce*, *primárny kľúč*, *obmedzenia*, *indexy*, atď. Podobne pre ostatné objekty. Každý databázový objekt, ktorý patrí do schémy má teda jednoznačne určeného predka – buď samotnú schému, alebo iný databázový objekt.

Stromová štruktúra každej databázovej schémy rešpektuje pravidlá rodičovského databázového stroja. Pre každý databázový stroj teda môžeme vytvoriť strom *typov databázových objektov*, z ktorého je zrejmé, akého predka bude mať konkrétna *inštancia databázového objektu*. Pre väčšinu databáz napríklad platí, že predok tabuľky je vždy samotná schéma, predok *stĺpca tabuľky* je objekt typu *tabuľka*, atď. Vzhľadom na to, že každý databázový stroj môže používať rozdielne databázové objekty, je prirodzené, že aj tieto pravidlá, a teda aj stromy typov jednotlivých databáz, sa môžu líšiť. Napríklad niektoré databázové stroje používajú *sekvencie* (DBMS Oracle, DBMS PostgreSQL), niektoré nie (DBMS MySQL). Pri porovnaní schém dvoch odlišných databázových strojov by vznikol problém, ako sa k takýmto objektom postaviť. Preto budeme musieť dopredu zadefinovať, ako bude vyzerať stromová štruktúra schémy a to pre každý uvažovaný databázový stroj samostatne. Navyše vytvoríme strom typov, ktorý budeme používať pri porovnaní schém z rôznych databázových strojov. Tento strom bude vychádzať z normy a bude obsahovať len spoločné typy, ktoré sa nachádzajú vo všetkých databázach akceptujúcich normu.

K typu databázového objektu sa jednoznačne viažu *vlastnosti objektu*. Vzhľadom na to, že budeme porovnávať objekty, ktoré môžu byť z rôznych databáz, budeme potrebovať presnejšie určenie typu, pretože napríklad vlastnosti objektu typu tabuľka môžu závisieť od

typu databázy. Tabuľka v DBMS Oracle môže mať vlastnosť *tablespace* (špecifikácia logického uloženia tabuľky v databáze), kým v MySQL táto vlastnosť nie je potrebná. Preto pre každý databázový objekt, ktorý sa nachádza v norme, zavedieme základný typ, napríklad typ *table*, a pre každý databázový stroj, ktorý budeme používať, odvodíme od základného typu potomka. V našom prípade *ora_table* pre DBMS Oracle, *my_table* pre DBMS MySQL, atď. Pre každý typ zavedieme množinu vlastností, pričom pre potomkov samozrejme doplníme len chýbajúce vlastnosti, pretože ostatné vlastnosti zdedia od predkov. Pri porovnávaní schém rôznych databázových strojov budeme používať základné objekty, pri porovnávaní dvoch schém toho istého databázového stroja zase objekty patriace tejto databáze.

3.4 Načítanie databázových schém do metaobjektov

V predchádzajúcej časti sme opísali, ako vyzerá databázová schéma. V tejto sekcii navrhneme spôsob vyžiadania informácií o schéme z databázy a spôsob ich uloženia.

Pri analýze sme zistili, že našim požiadavkám plne vyhovuje hotová aplikácia *Schemagic*, konkrétne jej časť, ktorá sa stará o získanie *metadát* z databázy a ich uloženie do XML súboru. Požiadavky boli nasledovné:

- možnosť definovať *typy databázových objektov* pre rôzne databázové stroje; *dedičnosť typov*
- možnosť definovať *stromy typov* pre rôzne databázové stroje
- informácie o schéme by sa mali dať získať buď zo systémových tabuliek, alebo z JDBC metadát; spôsob získania metadát musí byť maximálne konfigurovateľný
- výstup sa ukladá do XML súboru, kvôli zálohovaniu stavu schémy; neskôr sa môže použiť pri offline porovnaniach, aplikácia sa nebude pripájať priamo k databázam, ale bude analyzovať obsah XML súborov

Integrácii *Schemagicu* do našej aplikácie sa budeme venovať v kapitole popisujúcej samotnú implementáciu. Mapovanie výstupného XML súboru do pamäti je vďaka knižnici JAXB priamočiare, Java aplikácia teda môže pracovať s dvoma stromami objektov, ktoré reprezentujú referenčnú a porovnávanú databázovú schému.

3.5 Vyhľadanie ekvivalentov

Synchronizácia dvoch databázových schém je založená na hľadaní rozdielov medzi nimi. V prvom kroku sa ku každému objektu v *referenčnej databázovej schéme* snažíme nájsť jeho ekvivalent v *porovnávanej schéme*. Neznamená to, že hľadáme len identické

objekty, tj. také, ktoré majú rovnakú pozíciu v strome, rovnaký typ a rovnaké všetky vlastnosti. Naopak – pri hľadaní ekvivalentného databázového objektu porovnávame len kľúčové vlastnosti, ak sú rovnaké, prehlásime tieto objekty za zodpovedajúce si, resp. *ekvivalentné*. Dva databázové objekty (prvý z referenčnej, druhý z porovnávanej schémy) prehlásime za *ekvivalentné* vtedy a len vtedy, ak sú splnené nasledovné podmienky:

- typ obidvoch objektov je rovnaký (ak porovnávame schémy rovnakého databázoveho stroja); v prípade, že porovnávame schémy rôznych databázových strojov, musia mať typy objektov spoločného predka
- objekty sa zhodujú v kľúčových vlastnostiach
- predkovia obidvoch objektov sú navzájom ekvivalentné databázové objekty

Jedinú výnimku tvoria objekty, ktoré reprezentujú samotné schémy. Na začiatku prehlásime obidve schémy (bez ohľadu na ich vlastnosti) za ekvivalentné, pretože nás zaujíma, ako sa líšia objekty vo vnútri schém.

Algoritmus na vyhľadávanie ekvivalentov budeme realizovať prehľadávaním stromu do hĺbky, použijeme pritom rekurziu:

Algoritmus 1: Vyhľadávanie a označenie ekvivalentov

```
input:  reference schema refSchema
        comparison schema compSchema
output: reference schema and comparison schema with linked equivalent nodes

   Procedure compareChildren
   input:  reference object refObj
           comparison object compObj
   output: reference object and comparison object with linked equivalent children
   begin
      refLoop:
      for all refChild : refObj.children do
         compLoop:
         for all compChild : compObj.children do

            if compChild.equivalent != null then   { compChild already has an equivalent }
               continue compLoop
            fi

            if not typesCompatible(refChild, compChild) then  { types are not compatible }
               continue compLoop
            fi

            if keyPropertiesMatch(refChild, compChild) then   { key properties are equal }
               refChild.equivalent = compChild
               compChild.equivalent = refChild
               compareChildren(refChild, compChild)
               continue refLoop  { start searching for an equivalent of the next ref object }
            fi
         done
      done
   end

begin
   refSchema.equivalent = compSchema
   compSchema.equivalent = refSchema
   compareChildren(refSchema, compSchema)
end
```

V algoritme sú použité dve procedúry, ktoré kvôli prehľadnosti nie sú priamo uvedené v kóde. Prvá procedúra má názov *typesCompatible* a jej úlohou je otestovať, či sa typy porovnávaných objektov zhodujú. Ak boli obidve schémy, ktoré algoritmus spracováva, získané z rovnakého databázového stroja, procedúra vráti *true* vtedy a len vtedy, ak sa typy zhodujú, napr. keď sú obidva objekty typu *ora_table*. Ak schémy patria rôznym databázovým strojom, procedúra zisťuje, či majú typy spoločného predka; ak áno, vráti *true*,

v opačnom prípade vráti *false*. Napríklad typy *ora_table* a *my_table*, reprezentujúce tabuľky v DBMS Oracle a DBMS MySQL, majú spoločného predka – typ *table*. Metóda by teda aj v tomto prípade porovnávania objektov s uvedenými typmi vrátila *true*.

Druhou procedúrou, ktorej realizácia nie je uvedená v kóde, je procedúra *keyPropertiesMatch*. Jej úlohou je porovnať kľúčové vlastnosti obidvoch objektov. V čase jej vykonávania už vieme, že typy obidvoch objektov sú kompatibilné a poznáme prípadného spoločného predka obidvoch objektov. Podľa typu sa môžeme rozhodnúť, ktoré vlastnosti sú kľúčové pre porovnanie, postupne teda porovnáme ich obsah a v prípade, že sa všetky zhodujú, vrátime *true*. V opačnom prípade vrátime *false*. Implementácia tejto metódy bude zložitejšia, pretože nie vždy je jasné, ktoré vlastnosti sú kľúčové. Preto užívateľovi dáme možnosť rozhodnúť, ktoré vlastnosti sa budú porovnávať, v zložitejších prípadoch bude dokonca mať možnosť napísať skript v jazyku JavaScript. Bližšie informácie uvedieme v kapitole o implementácii.

3.6 Porovnanie ďalších vlastností

O ekvivalencii dvoch databázových objektov sme rozhodli na základe minimálneho počtu vlastností, ktoré tento objekt identifikujú – na základe *kľúčových vlastností*. Niektoré vlastnosti objektu však môžu byť rôzne. Napríklad k stĺpcu tabuľky referenčnej schémy môžeme nájsť rovnako pomenovaný stĺpec v rovnako pomenovanej tabuľke v porovnávanej schéme. Objekty preto prehlásime za ekvivalentné – zodpovedajúce si. Obidva stĺpce sa však nemusia zhodovať napríklad v dátovom type, čo už nie je kľúčová vlastnosť. Pri synchronizácii chceme upozorniť práve na tieto rozdiely, ktoré môže človek ľahko prehliadnuť. Naopak – niektoré vlastnosti, napriek tomu, že sú v dvoch zodpovedajúcich si objektoch rôzne, nebudeme chcieť vyznačiť ako zmenu. Typický príklad môže byť typ stĺpca pohľadu, ktorý je závislý na type zodpovedajúceho stĺpca tabuľky. Ak sa zmení typ stĺpca v tabuľke, automaticky sa zmení aj typ stĺpca v pohľade, pričom definícia pohľadu sa vôbec nezmenila a užívateľ nemusí robiť žiadne dodatočné kroky na aktualizáciu pohľadu.

Implementácia bude podobne ako pri porovnávaní ekvivalentov zložitejšia. Užívateľ bude mať možnosť vybrať zoznam vlastností, ktoré sa budú porovnávať, alebo bude môcť napísať zložitejší skript. Napríklad meno objektu sa bude porovnávať len vtedy, ak nebolo automaticky vygenerované databázovým systémom. Porovnávanie sa bude vykonávať hneď po označení ekvivalentov v procedúre *compareChildren*, nebudeme teda musieť opätovne prechádzať obidva stromy. Pre každú dvojicu ekvivalentných – tj. zodpovedajúcich si –

objektov si budeme pamätať zoznam zmenených vlastností a ich hodnoty v referenčnom aj v porovnávanom objekte. Detaily implementácie budú opäť uvedené v samostatnej kapitole.

3.7 Zlúčenie stromov do diff stromu

Výstupom predchádzajúcich algoritmov sú vzájomné odkazy medzi *metaobjektami* reprezentujúcimi reálne databázové objekty referenčnej a porovnávanej schémy. Po vykonaní algoritmov máme stále k dispozícii dva stromy, reprezentované dvoma koreňami. Strom referenčnej schémy na jednej strane a strom porovnávanej schémy na strane druhej. Medzi objektami týchto stromov však vedú spojenia označujúce ekvivalenciu. Pre každé takéto spojenie existuje zoznam vlastností, ktorých hodnota je rôzna v referenčnom a v porovnávanom objekte. Ak existuje spojenie, ale zoznam zmenených vlastností je prázdny, znamená to, že objekty sa nijako nelíšia a sú rovnaké.

V prípade, že sa k objektu nenašiel žiadny ekvivalentný objekt v druhej schéme, znamená to, že je nový, alebo bol zmazaný, podľa toho, v akom strome sa nachádza, alebo bola zmenená niektorá z jeho kľúčových vlastností. Postupne rozoberieme všetky prípady.

Najprv si pripomenieme, ako sa líšia dve uvažované databázové schémy. Užívateľ ako referenčnú schému vyberie tú, ktorú považuje za cieľovú. Tj. chce zistiť, ako sa *porovnávaná* databázová schéma líši od *referenčnej*. Ak teda existuje objekt v strome reprezentujúcom referenčnú schému, ktorý nemá ekvivalentný objekt v porovnávanej schéme, jedná sa o *nový* objekt, tj. objekt, ktorý pribudol. Ak naopak existuje objekt v strome porovnávanej schémy, ktorý nemá ekvivalent v referenčnej schéme, jedná sa o tzv. *zastaralý* objekt, tj. objekt, ktorý už v referenčnej schéme nie je. Zhrnutie je uvedené v nasledujúcej tabuľke:

Stav objektu	**Objekt je v schéme**	**Podmienky**
žiadna zmena (*no difference*)	referenčná, porovnávaná	objekt má ekvivalent, zoznam zmenených vlastností je prázdny
zmenené vlastnosti (*changed*)	referenčná, porovnávaná	objekt má ekvivalent, zoznam zmenených vlastností nie je prázdny
nový objekt (*new*)	referenčná	objekt nemá ekvivalent, nachádza sa v referenčnej schéme
zastaralý objekt (*obsolete*)	porovnávaná	objekt nemá ekvivalent, nachádza sa v porovnávanej schéme

Tabuľka 1: Možné stavy databázového objektu

Napriek tomu, že už vieme rozhodnúť, aký stav treba priradiť databázovému objektu, stále nemáme k dispozícii dátovú štruktúru, ktorú by sme mohli prechádzať a generovať výstup. Na to, aby sme ju získali, musíme zliať (merge) stromy reprezentujúce obidve

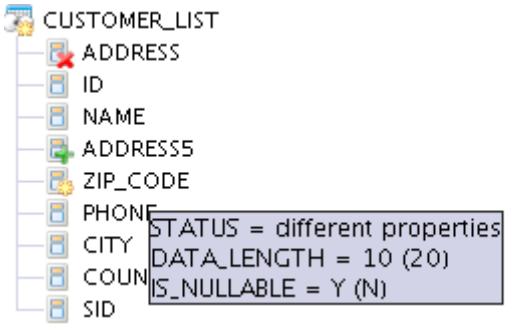
schémy do jedného *rozdielového stromu*, ktorý budeme nazývať aj *diff strom*. Diff strom bude obsahovať všetky zmenené aj nezmenené databázové objekty. Ako základ pre jeho vytvorenie bude slúžiť referenčný strom, do neho budeme postupne vkladať zastaralé (obsolete) objekty zo stromu porovnávanej schémy. Objekty, ktoré majú v referenčnom strome ekvivalent neprenášame, pretože by sa vo výslednom strome vyskytovali dvakrát.

Algoritmus 2: Označenie stavu objektu a vytvorenie diff stromu

```
input:  reference schema tree root refSchema { with linked equivalent nodes }
        comparison schema tree root compSchema { with linked equivalent nodes }
output: diff tree root

  Procedure postProcessTree
  input:  database object obj
          type of tree treeType
  output: obj and all its children have the state property filled
  begin
    if obj.equivalent != null then
      if obj.changedProperties is empty then
        obj.state = NO_DIFFERENCE
      else
        obj.state = CHANGED
      fi
    else
      if treeType is REFERENCE_TREE then  { reference tree}
        obj.state = NEW
      else   { comparison tree }
        obj.state = OBSOLETE
      fi
    fi

    for all child : obj.children do
      postProcessTree(child, treeType)
    done
  end

  Procedure mergeCompToRef
  input:  comparison object compObj
          parent of comparison object compParent
  output: objects with no equivalent from comparison tree merged to the reference tree
  begin
    if compObj.equivalent != null then      { compObj has an equivalent }
    for all child : compObj.children do
        mergeCompToRef (child, compObj)
      done
    else   { compObj does not have an equivalent }
      refParent = compParent.equivalent
      pos = position of the first refParent's child of the same type or -1 if not found
```

```
            if pos == -1 then
                refParent.children.add(compObj)
            else
                {insert before all other children that have the same type }
                refParent.children.insert(pos, compObj)
            fi
        fi
    end

begin
    postProcessTree(refSchema, REFERENCE_TREE)
    postProcessTree(compSchema, COMPARISON_TREE)
    mergeCompToRef(compSchema, null)
    diff tree root = refSchema
end
```

Algoritmus najprv spracuje každý uzol obidvoch stromov zavolaním metódy *postProcessTree*. V prípade stromu referenčnej schémy označí každý uzol bez ekvivalentu ako *new*, v prípade stromu porovnávanej schémy označí takýto uzol ako *obsolete*. Všetky uzly teda majú jednoznačne pridelený stav (state). Ako základ diff stromu algoritmus použije strom referenčnej schémy. Postupne prechádza uzly druhého stromu a ak nájde uzol bez ekvivalentu (takýto uzol má určite nastavený stav *obsolete*), pripojí ho aj s celým podstromom do referenčného stromu. Nový rodičovský uzol bude tvoriť ekvivalent pôvodného rodiča. Uzol sa zaradí pred všetky uzly rovnakého typu, to znamená, že bude zaručené, že všetky *obsolete* objekty sú zaradené pred *new* objektami toho istého typu. Ak napríklad porovnávaná schéma vznikla kópiou referenčnej schémy, pričom neskôr niekto v referenčnej schéme z tabuľky T_1 vymazal stĺpec C_1, metaobjekt stĺpca C_1 sa bude nachádzať v strome porovnávanej schémy, ale nie v strome referenčnej schémy. Pre tento objekt sa teda nenájde ekvivalent a algoritmus ho označí ako *obsolete*. Pri zlievaní stromov algoritmus najprv nájde nového rodiča – metaobjekt reprezentujúci tabuľku T_1 v referenčnej schéme. Následne metaobjekt stĺpca C_1 zaradí na prvé miesto, pred všetky ostatné metaobjekty reprezentujúce nezmenené, alebo nové stĺpce.



Obr.: Diff strom reprezentovaný v GUI

Na obrázku je zachytená časť diff stromu, ktorý je reprezentovaný prostredníctvom vytvoreného GUI. Pohľad CUSTOMER_LIST má nastavený stav na *changed*, pretože niektoré jeho vlastnosti sa zmenili. Stĺpec ADDRESS je *obsolete*, pretože sa nachádzal len v strome porovnávanej schémy a nie v strome referenčnej schémy. Je zaradený pred všetkými ostatnými. Stĺpce bez akéhokoľvek znaku sú bez zmeny, tj. bol k nim nájdený ekvivalent a všetky porovnávané vlastnosti sa zhodujú. Stĺpec ADDRESS5 má stav *new*, pretože bol len v strome referenčnej schémy a nie v strome porovnávanej schémy. Napokon stĺpec ZIP_CODE má stav *changed*. Je v obidvoch stromoch, kľúčová vlastnosť (meno) sa zhoduje, takže objekty sú ekvivalentné. Ďalšie porovnávané vlastnosti ale majú rôzne hodnoty – konkrétne ide o vlastnosť DATA_LENGTH, kde referenčný objekt má nastavenú hodnotu 10 a hodnota tejto vlastnosti v porovnávanom objekte je 20. Podobne vlastnosť IS_NULLABLE má v prvom prípade hodnotu Y, v druhom hodnotu N.

3.8 Spracovanie diff stromu

Diff strom je výstup porovnávacej časti aplikácie. Z hotového diff stromu môžeme generovať požadovaný výstup pre užívateľa – či už pôjde o XML súbor, textový výstup, alebo o SQL skript, ktorý modifikuje porovnávanú schému tak, aby zodpovedala referenčnej schéme.

Pri generovaní textu, prípadne XML výstupu, prechádzame diff strom do hĺbky a pre každý objekt vygenerujeme zodpovedajúci výstup. Napríklad pre uzol, ktorého stav je *unchanged* vygenerujeme len základné informácie, pre uzol, ktorého stav je *changed* vypíšeme názvy zmenených vlastností a ich hodnoty, atď. Generovanie SQL výstupu je zložitejší problém, ktorý podrobnejšie rozoberieme v ďalšej kapitole, princíp prechádzania diff stromu a reagovanie na stav objektu a jeho vlastnosti je však podobný aj v prípade generovania SQL kódu.

Algoritmus 3: Spracovanie diff stromu

input: diff tree root *diffRoot*
output: text information about differences between schemas

```
Procedure process
input:  diff object diffObj
output: diffObj and all its children processed
begin
   print diffObj.name, diffObj.state

   if diffObj.state == CHANGED then
   for all changedProp : diffObj.changedProperties do
      print changedProp.name, changedProp.refValue, changedProp.compValue
      done
   fi

   for all child : diffObj.children do
      process (child)
   done

end

begin
   process(diffRoot)
end
```

Algoritmus 3 prechádza diff strom a pre každý uzol vypíše základné informácie (meno databázového objektu a jeho stav). Pre objekty, ktorých stav je nastavený na *changed* (tj. hodnoty vlastností sa líšia) vypíše aj názvy zmenených vlastností, spolu s obidvoma hodnotami. Textový výstup je najjednoduchší z troch druhov výstupov, ktoré budeme implementovať.

3.9 Generovanie SQL príkazov

Ako sme už naznačili, aj pri generovaní najzložitejšieho typu výstupu, tj. pri generovaní SQL príkazov, budeme používať prechádzanie diff stromu do hĺbky. Budeme teda používať algoritmus podobný Algoritmu 3. Nevystačíme si však s jednoduchým návratom textového reťazca. Pri generovaní SQL kódu potrebujeme reagovať na stav a vlastnosti jednotlivých objektov. Podľa nich vygenerujeme SQL kód pre príslušnú databázu, tj. databázu, v ktorej sa nachádza porovnávaná schéma.

Podoba SQL kódu môže závisieť na predkoch, aj potomkoch aktuálneho uzla. Napríklad ak pracujeme s uzlom, ktorý reprezentuje tabuľku, jej stav je *new* a uzol obsahuje

niekoľko poduzlov, ktoré reprezentujú stĺpce tabuľky, pre všetky tieto uzly vytvoríme len jeden príkaz *CREATE TABLE*, ktorý bude zahŕňať aj vytvorenie stĺpcov. Uzol tabuľky ale môže obsahovať aj ďalšie poduzly (obmedzenia, indexy), na ktoré už budeme reagovať samostatne. Podobná situácia nastane pre uzol, ktorý reprezentuje tabuľku v stave *obsolete*. Napriek tomu, že uzol môže obsahovať mnoho poduzlov, väčšina z nich bude automaticky vymazaná databázovým strojom pri zavolaní príkazu *DROP TABLE*. Systém teda musí umožniť obsluhu poduzla z hlavného uzla.

Práve kvôli komplexnosti tohto problému do systému zavedieme tzv. *pravidlá* (rules), ktoré budú určovať, aký SQL kód sa má generovať pre objekt. Každý objekt bude mať svoje pravidlá, tie sa navyše budú deliť podľa toho, v akom stave objekt je. Má zmysel reagovať na tri stavy, ktoré znamenajú zmenu – ak je objekt *nový* (new), ak je *zastaralý* (obsolete), alebo ak má *zmenené niektoré vlastnosti* (changed). Užívateľ môže napríklad vytvoriť pravidlá, ktoré sa budú vykonávať, ak algoritmus v strome nájde uzol, ktorý reprezentuje tabuľku v stave *new*, alebo index v stave *changed*, atď. Ku každému typu objektu a stavu môže existovať nula až neobmedzene veľa zoradených pravidiel, použitie každého z nich bude obmedzené podmienkou. Vykoná sa vždy prvé pravidlo, ktorého podmienka bude splnená.

Každé pravidlo obsahuje sekvenciu príkazov, ktoré sa po vybratí pravidla vykonajú. Každý príkaz vygeneruje časť textu výsledného SQL príkazu, ktorý zabezpečí zmeny spracovávaného objektu v porovnávanej schéme. V príkazoch bude možnosť používať premenné, ktoré sa v čase vykonávania nahradia skutočnou hodnotou. Napríklad pravidlo, ktoré bude priradené typu tabuľka v stave *obsolete* bude obsahovať len jeden príkaz na generovanie textu, popísaný predpisom *DROP TABLE {table.name}*, kde poslednú časť príkazu tvorí zápis premennej, ktorá sa nahradí hodnotou vlastnosti *name* objektu tabuľka.

Každý príkaz v pravidle má (podobne ako samotné pravidlo) vstupnú podmienku. Príkaz je vykonaný vtedy a len vtedy, ak je podmienka splnená. Vykonanie príkazu znamená vrátenie textového reťazca, ten sa generuje podľa predpisu, ktorý príkaz obsahuje. Predpis je kombinácia pevného textu a premenných, ktoré sa musia nahradiť hodnotou platnou v čase vykonávania pravidla.

Pravidlo môže obsahovať aj tzv. *cykly* (loops). Cykly nám umožňujú obsluhu poduzlov, napríklad už spomenutý prípad vytvorenia novej tabuľky, kde chceme do jedného príkazu *CREATE TABLE* umiestniť aj všetky definície nových stĺpcov, vyriešime práve pomocou nich. Cyklus je platný vždy pre všetkých potomkov daného typu – napríklad pre stĺpce tabuľky. Každý cyklus opäť obsahuje sekvenciu príkazov, alebo ďalšie cykly. Pretože

príkazy môžu obsahovať podmienky, môžeme niektoré iterácie efektívne „preskočiť“ nastavením vhodnej podmienky – napríklad môžeme do príkazu *CREATE TABLE* zaradiť len stĺpce, ktoré sú súčasťou primárneho kľúča.

Keďže pri generovaní SQL postupujeme podľa Algoritmu 3, platí, že SQL sa tvorí postupne a pozícia SQL príkazu obsluhujúceho zmeny konkrétneho objektu sú jednoznačne dané pozíciou objektu v diff strome. Toto nie je vždy želaná vlastnosť, pretože sa môže stať, že príkaz sa odvoláva na objekt, ktorý ešte nevznikol, resp. nebol modifikovaný. Napríklad cudzí kľúč sa môže odvolávať na novovzniknutú tabuľku, ktorej vytvárací príkaz bude v skripte až za vytváracím príkazom cudzieho kľúča, čo pri neskoršom vykonávaní skriptu spôsobí chybu. Tejto chybe sa dá jednoducho predísť odložením vytváracieho príkazu cudzieho kľúča na koniec SQL skriptu. Keďže takúto funkčnosť budeme potrebovať obecne, v každom pravidle bude užívateľ môcť vytvoriť samostatnú sekvenciu príkazov a cyklov, ktorých výsledky budú určené na iné, než štandardné miesto v SQL skripte. Každý príkaz, alebo cyklus v tejto sekvencii bude mať oproti bežným príkazom a cyklom navyše jeden parameter – *skupinu SQL príkazov* v rámci ktorej sa vykoná. V globálnych nastaveniach generátora sa určí postupnosť skupín. Postupnosť skupín môže vyzerať napríklad takto: dropTriggers, dropConstraints, [default], addFks. To znamená, že najprv budú v SQL skripte vystupovať texty, ktoré vznikli spustením príkazov s definovanou skupinou dropTriggers, nasledovať bude skupina dropConstraints. Do skriptu sa potom vložia všetky príkazy, ktoré nemajú určenú skupinu (štandardné príkazy). Na koniec SQL skriptu sa vložia už spomínané cudzie kľúče, chyba pri vykonávaní teda nenastane.

Pri generovaní SQL je potrebné riešiť aj ďalšie problémy. Napríklad dátový typ v DBMS Oracle je daný niekoľkými vlastnosťami – *data_type*, *data_scale*, *data_length*, *data_precision*. Stačí, že len jedna z týchto vlastností je zmenená a predpis dátového typu sa musí zmeniť. Problém je, že ho nemôžeme „vygenerovať“ napríklad jednoduchým spojením textov vlastností za seba. Niektoré dátové typy napríklad vo svojom popise využívajú vlastnosť *data_length* (napr. VARCHAR2(10)), iné nie (napr. TIMESTAMP). Na to, aby sme správne vygenerovali predpis dátového typu teda budeme potrebovať špeciálne pravidlá, ktoré podľa názvu typu rozhodnú, aké parametre dátový typ potrebuje a aké vlastnosti sú na to potrebné. Rovnako konverzie dátových typov medzi databázovými strojmi, alebo tzv. *util* funkcie, ako nahradzovanie znakov inými znakmi, budeme pri generovaní SQL nutne potrebovať. Nebolo by zrejme účelné vytvárať samostatné pravidlá aj na takéto drobné úlohy, preto si v implementácii pomôžeme skriptami jazyka JavaScript. V nich si užívateľ bude môcť vytvoriť vlastné užívateľské funkcie, ktoré potom využije pri

generovaní SQL. Funkcia napríklad dostane všetky parametre, ktoré súvisia s dátovým typom a vygeneruje textový predpis dátového typu. Výsledok volania sa bude do príkazov v pravidlách vkladať podobne, ako premenné. Užívateľ bude mať k dispozícii štandardnú knižnicu JavaScript funkcií, ktorú bude môcť upravovať a dopĺňať.

4 POUŽITÉ TECHNOLÓGIE

Pri tvorbe aplikácie budeme používať niekoľko existujúcich technológii. Pre túto prácu sú ťažiskové najmä *databázové technológie*, v tejto kapitole ale uvedieme aj ostatné technológie, ktoré neskôr použijeme.

4.1 Podporované databázy

Hlavná databáza, pre ktorú bude synchronizačná aplikácia vytvorená je *DBMS Oracle*. Podporované sú aj relačné databázy od iných dodávateľov, ku ktorým existuje *JDBC driver*. Ako príklad použitia alternatívneho databázového riešenia budeme používať *DBMS MySQL*.

4.1.1 SQL

Pri porovnávaní schém dvoch odlišných databáz budeme vychádzať z už spomínanej normy *SQL-92*. Pri generovaní výstupnej sekvencie SQL príkazov môžeme využiť natívny dialekt danej databázy. V našom prípade napríklad konštrukty jazyka *PL/SQL*, ktorý používa *DBMS Oracle*. Tento výstup bude plne konfigurovateľný a užívateľ si bude môcť sám upraviť preddefinované nastavenia.

4.2 Java

Porovnávanie databázových schém bude realizované v jazyku Java vo verzii J2SE 5.0. Jazyk Java je multiplatformový, aplikácia teda bude fungovať na všetkých operačných systémoch, pre ktoré existuje interpreter Javy. Veľkou výhodou tohto programovacieho prostredia je overená technológia *JDBC*, spoločná vrstva na prístup k rôznym databázam. Java ponúka veľké množstvo knižničných funkcií, bezproblémovú prácu s vláknami, ako aj nástroje na tvorbu grafického rozhrania založeného na báze princípu *Model-View-Controller*. Ďalšou výhodou je existencia množstva voľne dostupných externých knižníc, ktoré implementujú použité technológie a integrácia aplikácie s týmito knižnicami je

priamočiara. My budeme potrebovať hlavne knižnice na spracovanie XML dokumentov. Dôležitým argumentom bolo aj to, že nástroj *Schemagic* je vytvorený v jazyku Java, pri použití rovnakého jazyka sa teda vyhneme problémom s prepojením obidvoch aplikácií.

4.2.1 JDBC

JDBC (Java Database Connectivity) je štandard na prepojenie kódu napísaného v Jave s databázami – spravidla s relačnými databázami, ale aj s ostatnými dátovými zdrojmi, ako napríklad so súbormi tabuľkových procesorov, alebo s textovými súbormi [12]. Programátor pracuje s JDBC vrstvou pomocou sady funkcií definovaných v štandarde JDBC API. V podstate k akémukoľvek dátovému zdroju, pre ktorý existuje tzv. *JDBC driver* môže pristupovať použitím jednotnej sady týchto funkcií. JDBC driver poskytuje aj tzv. *metadata* – podrobné informácie o databáze, ktorú driver obsluhuje. Pomocou týchto dát napríklad môžeme zistiť, aké štandardy databázový stroj podporuje, aké databázové objekty môže obsahovať, alebo aké dátové typy môže užívateľ používať.

Napriek tomu, že JDBC je dobre navrhnutá vrstva na spoločný prístup k databázam, budeme ju využívať len čiastočne a to na pripájanie k dátovým zdrojom. Oveľa menej budeme využívať metadáta, ktoré sú príliš závislé na implementácii JDBC drivera. Programátor sa môže spravidla plne spoľahnúť na to, že JDBC driver správne sprostredkuje volanie do databázy a po interpretácii SQL príkazu databázovým strojom vráti správny výsledok. Už menej sa môže spoľahnúť na to, že metadata sú korektné a hlavne úplné. Napríklad oficiálny JDBC driver k DBMS PostgreSQL do zoznamu tabuliek umiestňuje všetky objekty (napr. indexy), ktoré by tam podľa JDBC normy vôbec nemali byť. Programátor musí poznať toto správanie konkrétneho JDBC drivera, aby naň mohol reagovať. Keďže cieľom je vytvoriť aplikáciu, do ktorej môže sám užívateľ pridať dátový zdroj, správanie aplikácie by nemalo závisieť na vlastnostiach drivera. Na získanie dát z databázy preto použijeme nástroj *Schemagic*, v ktorom sa informácie o štruktúre databázovej schémy dajú získať buď priamo, pomocu dotazov do systémových tabuliek, alebo (ak tieto informácie interpretuje JDBC driver korektne) z metadát.

4.2.2 JFC/Swing

Napriek tomu, že jednou z požiadaviek je, aby bola aplikácia konzolová a dala sa spúšťať automatizovane, budeme potrebovať aj jej grafickú verziu (tzv. *GUI* – Graphical User Interface), v ktorej bude môcť užívateľ upraviť konfiguračné súbory. Priama editácia XML súborov by bola v tomto prípade neúmerne zdĺhavá a neprehľadná.

Keďže sme sa už rozhodli aplikáciu implementovať v jazyku *Java*, zostáva vybrať vhodnú knižnicu na implementáciu jej grafickej časti. Možností je niekoľko – môžeme vytvoriť webovú aplikáciu pomocou niektorého z množstva existujúcich *frameworkov*, alebo použiť niektorú z knižníc, ktorá umožňuje tvorbu štandardných „okien“ (pre Javu existujú dve hlavné knižnice – *JFC/Swing* a *SWT*). Keďže *JFC/Swing* je knižnica vstavaná priamo do *J2SE* a má prepracovaný systém komponent založený na architektúre *Model-View-Controller*, rozhodli sme sa ju použiť. Synchronizácia teda bude mať svoje grafické rozhranie, bude sa dať spustiť priamo z tohto rozhrania a užívateľ bude môcť GUI využiť na podrobné nastavenie synchronizácie, aj na prehľadné zobrazenie jej výsledkov.

4.3 JavaScript

Synchronizačná aplikácia by mala byť dostatočne jednoduchá pre bežného používateľa, ale aj dostatočne flexibilná pre skúsených užívateľov. Každú úlohu, či už pôjde o porovnávanie databázových objektov, alebo o generovanie SQL kódu, bude užívateľ môcť riešiť dvoma spôsobmi – štandardným, kde pomocou komponent v GUI zvolí želané nastavenie, alebo programovým kódom v jazyku *JavaScript*. Týmito skriptami bude môcť výrazne ovplyvniť štandardnú funkcionalitu aplikácie.

JavaScript je skriptovací jazyk, ktorý pôvodne vyvinula firma Netscape. Netscape neskôr nechal jazyk štandardizovať, túto štandardizovanú verziu dnes poznáme pod názvom *ECMAScript* [13]. Keďže porovnávanie aj generovanie SQL sekvencie bude prebiehať v jazyku Java, budeme musieť integrovať tieto dve technológie – na úrovni premenných – v obidvoch jazykoch. Tj. potrebujeme „exportovať“ niektoré objekty z jazyka Java do JavaScriptu. Naopak – komplexné výsledky budeme späť do Javy vracať zapísaním do exportovaných objektov. Jednoduchšie výsledky – textový reťazec, číslo, atď. – vrátime štandardným príkazom jazyka JavaScript *return*.

Na bezproblémovú integráciu Javy a JavaScriptu použijeme knižnicu *Rhino*. *Rhino* je open-source implementácia JavaScriptu, celá je napísaná v Jave. Typicky sa používa v Java aplikáciách a umožňuje koncovým užívateľom používať skriptovací jazyk [14].

4.4 XML

XML (Extensible Markup Language) je špecifikácia, podľa ktorej sa dajú vytvárať vlastné jazyky založené na špeciálnych značkách, tzv. *tagoch* [15]. My budeme XML využívať pomerne často. Bude nám slúžiť ako formát na uloženie konfiguračných súborov, napríklad na uloženie parametrov synchronizácie, alebo na uloženie pravidiel pre

generovanie sekvencie SQL príkazov. XML formát budeme používať aj na integráciu programu *Schemagic* do našej aplikácie na porovnávanie databázových schém. Budeme totiž používať výstup *Schemagicu*, ktorý je vždy uložený práve vo formáte XML.

Pre jazyk *Java* existuje niekoľko knižníc, ktoré zjednodušujú prácu programátora pri manipulovaní s XML súbormi. Na prácu s jednoduchšími XML súbormi – väčšinou sa bude jednať o konfiguračné súbory grafického prostredia – budeme používať knižnicu *JDOM* [16]. Na spracovanie zložitejších XML súborov (spravidla konfiguračných súborov synchronizácie a generovania SQL) budeme používať knižnicu *JAXB* [17].

4.4.1 XML Schema

XML Schema [18] je jazyk, pomocou ktorého sa dá definovať štruktúra a vlastnosti XML dokumentu. Pokiaľ XML dokument deklaruje, že dodržiava pravidlá definované danou schémou, označíme ho ako *inštanciu* danej schémy v jeho hlavičke. Schéma určuje štruktúru značiek. Definuje koreň a pre každú značku definuje zoznam nasledovníkov a obmedzenie ich počtu. Rovnako určuje aj dátové typy, ktoré sú prípustné v atribútoch. Atribút môže napríklad obsahovať textový reťazec, alebo číslo. Pre všetky XML dokumenty, ktoré nastavujú správanie synchronizačnej aplikácie, ale aj pre XML dokumenty, ktoré obsahujú dáta získané z databázy, existuje samostatná schéma. Zoznam použitých XML schém nájde čitateľ na priloženom CD.

4.5 Schemagic

Ako sme už naznačili, metadáta získané pomocou JDBC vrstvy nie sú pre účely tejto práce dostatočné, najmä kvôli malému množstvu informácií a kvôli ich nekonzistentnosti. Databázové systémy väčšinou poskytujú informácie o databázových objektoch v tzv. systémových tabuľkách. Napríklad v DBMS Oracle získame zoznam tabuliek dotazom do systémovej tabuľky `ALL_TABLES`. V dotaze môžeme použiť aj zložitejšie podmienky, napríklad môžeme požadovať len tabuľky s konkrétnym menom, alebo tabuľky, ktoré vlastní zvolený užívateľ. Tieto informácie sú väčšinou viazané ku konkrétnemu databázovému stroju. Každý typ databázy ma iné systémové tabuľky a v nich uložené informácie. Napríklad údaje o dátových typoch nie sú „prenositeľné“, tj. nebudú platiť v inom type databázy. Ak teda chceme získavať informácie o štruktúre databáz, musíme:

- vytvoriť spoločné „metaobjekty“, tj. objekty, na ktoré sa budú mapovať informácie o štruktúre bez ohľadu na typ databázy
- vytvoriť SQL dotazy, ktoré získajú tieto informácie z databázy

- vytvoriť mapovanie medzi výsledkami SQL dotazov a metaobjektami

Inštancie metaobjektov budú potom tvoriť vstup do synchronizačnej aplikácie.

Keďže systém, ktorý tieto úlohy spĺňa už existuje, rozhodli sme sa nevyvíjať vlastný, ale použiť existujúci. Systém *Schemagic* [7] vznikol s rovnakým cieľom, ako táto diplomová práca. K databázam sa pripája prostredníctvom JDBC vrstvy, na získavanie metadát však v prevažnej miere využíva systémové tabuľky. Konfiguruje sa XML súbormi. Tieto konfiguračné súbory sú prehľadné a plne vyhovujú požiadavkám, ktoré sa kladú na našu aplikáciu. Výstup zo *Schemagicu* tvorí opäť XML súbor, ktorý popíšeme v ďalších kapitolách.

5 IMPLEMENTÁCIA

V predchádzajúcej kapitole, ktorá sa venovala analýze, sme podrobne rozobrali problémy, ktorými sa táto práca zaoberá. Navrhli sme riešenia, niektoré detaily sme však zámerne neuviedli. Niekoľkokrát sme napríklad spomenuli použitie skriptovacieho jazyka JavaScript, neuviedli sme ale, k akým konkrétnym objektom má tento jazyk prístup a akým spôsobom môže vrátiť výsledné hodnoty. Popísali sme dátové štruktúry, z ktorých bude aplikácia získavať údaje o schémach, ale nevenovali sme sa XML súborom, z ktorých sa tieto dátové štruktúry vytvoria. Podobné tvrdenie platí pre konfiguráciu aplikácie – hovorili sme napr. o pravidlách na generovanie SQL, presnú definíciu pravidiel sme zatiaľ neuviedli. Kapitola o implementácii sa bude venovať práve týmto problémom: integrácii JavaScriptu do aplikácie, spôsobu použitia XML súborov a popisu ďalších implementačných detailov aplikácie na synchronizáciu databázových schém.

5.1 Vstup aplikácie

Vstup aplikácie budú tvoriť tri súbory. Prvé dva XML súbory budú popisovať referenčnú a porovnávanú schému, v každom z týchto súborov budú uložené informácie o strome databázových objektov v databázovej schéme. Z týchto informácii neskôr v jazyku Java vytvoríme stromy metaobjektov. Tieto dva vstupné súbory získame použitím aplikácie *Schemagic*. Tretí vstupný súbor bude konfiguračný, budú v ňom napríklad pravidlá, podľa ktorých sa rozhodne, či sú dva databázové objekty ekvivalentné, alebo pravidlá generovania SQL skriptu. Konfiguračný súbor sa dá vytvoriť ručne, alebo použitím GUI nástroja. GUI nástroju sa budeme venovať v samostatnej kapitole.

5.1.1 Schemagic

Nástroj *Schemagic* [7] slúži na získavanie informácii o ľubovoľnej databázovej schéme a dá sa čiastočne použiť aj na synchronizáciu databázových schém. Pre nás bude dôležitý hlavne modul *Schemagicu*, ktorý do XML súboru uloží strom reprezentujúci zvolenú

databázovú schému. Tento modul je konfigurovateľný a môžeme ho použiť pre akúkoľvek databázu, ku ktorej existuje JDBC driver.

5.1.2 Konfigurácia nástroja Schemagic

Prvým krokom konfigurácie je vytvorenie definície typov databázových objektov a vzťahov medzi nimi pre každý databázový stroj, ktorý budeme používať. Nástroj *Schemagic* má v projekte vlastný adresár (`schemagic/`), akékoľvek užívateľské definície sú v jeho podadresári `def/`. Konfiguračné súbory typov databázových objektov nájdeme v ďalšom podadresári `object/`, konfiguračné súbory vzťahov medzi objektami a pravidlá na získanie metainformácii v podadresári `machine/`. V tejto kapitole popíšeme najčastejšie používané konfigurácie, systém *Schemagic* však ponúka množstvo spôsobov na ďalšie upravenie výsledkov. Podrobné informácie o typoch konfigurácie nájde čitateľ v zodpovedajúcej práci [7]. Tento dokument je k dispozícii aj na priloženom CD.

5.1.2.1 Databázové objekty

Zoznam typov databázových objektov a ich vlastností je štandardne uložený v súbore `schemagic/def/object/<názov databázového stroja>_objects.xml`. Napríklad pre DBMS Oracle existuje súbor `oracle_objects.xml`. Definícia typu obsahuje vlastnosti, ktoré nás pri tomto type zaujímajú. Pretože veľa databázových objektov je spoločných a nájdeme ich vo všetkých uvažovaných databázových strojoch, existuje ešte súbor `objects.xml`, ktorý obsahuje predkov systémovo špecifických databázových objektov. Keďže definície typov podporujú dedičnosť, predkovia obsahujú spoločné vlastnosti (také, ktoré vychádzajú z normy), špecifické typy databázových strojov tieto vlastnosti dedia a pridávajú vlastné. Ukážeme si príklad definície typu *tabuľka* (table).

```
<object-def type="table">
  <properties-def>
  <property-def name="NAME" key="true"/>
  <property-def name="ROWS_COUNT"/>
  </properties-def>
</object-def>
```

Príklad: Definícia typu *table* v súbore `objects.xml`

```
<object-def type="ora_table">
  <extends type="table"/>
  <properties-def>
    <property-def name="TABLESPACE_NAME"/>
    <property-def name="CLUSTER_NAME"/>
    <property-def name="USRCOMMENT"/>
  </properties-def>
</object-def>
```

Príklad: Definícia typu *ora_table* v súbore `oracle_objects.xml`

V definícii základného typu sa vyskytujú dve vlastnosti – NAME (názov tabuľky) a ROWS_COUNT (počet riadkov tabuľky). Ak sa jedná o tabuľku v DBMS Oracle, okrem týchto vlastností nás zaujímajú aj ďalšie tri vlastnosti (TABLESPACE_NAME, CLUSTER_NAME a USRCOMMENT). Preto v súbore `oracle_objects.xml` vytvoríme špeciálny typ *ora_table*, zapíšeme do neho dedičnosť zo základného typu (`<extends type="table">`) a do definície vložíme špecifické vlastnosti.

Pripúšťame aj existenciu objektov, ktoré nemajú predka medzi základnými objektmi. Napríklad typ *ora_procedure*, ktorý reprezentuje uložené procedúry v DBMS Oracle je definovaný v súbore `oracle_objects.xml` a nemá žiadneho predka.

V ďalšom texte uvidíme, že ak budeme porovnávať dve schémy z DMBS Oracle, budeme mať pri spracovaní objektu typu tabuľka k dispozícii základné, aj systémovo špecifické vlastnosti. Ak bude jedna schéma patriť DBMS Oracle a druhá inému podporovanému databázovému systému, pri vytváraní porovnávacích pravidiel budeme môcť používať len základné vlastnosti.

5.1.2.2 Štruktúra databázovej schémy

V kapitole 3.3 sme podrobne popísali, akú štruktúru má databázová schéma. Databázové objekty sú v nej zoradené hierarchicky, napríklad stĺpce tabuľky sú vždy podriadené tabuľke, ktorá je zas podriadená rodičovskej schéme. Táto hierarchia je dodržaná v každej inštancii databázovej schémy konkrétneho databázového stroja, tj. môžeme ju definovať na úrovni typov databázových objektov. Opäť platí, že bude existovať spoločná hierarchia, ktorá obsahuje len objekty definované normou a pre každý databázový stroj môže existovať vlastná hierarchia, obsahujúca aj systémovo špecifické objekty. V prípade hierarchií nepoužívame dedičnosť, tj. každú hierarchiu vytvárame odznovu a nepoužívame spoločný základ. Mierne predbehneme text a už na tomto mieste naznačíme,

že pri porovnávaní dvoch schém rovnakého DBMS sa použije hierarchia definovaná pre tento databázový stroj (ak taká hierarchia existuje), pri porovnávaní schém dvoch rôznych DBMS sa použije základná hierarchia.

Základná hierarchia je definovaná v adresári `schemagic/def/machine/`, súbor `generic.machine.xml`, systémovo špecifická hierarchia v súbore `schemagic/def/machine/<názov databázového stroja>.machine.xml`. V prvej sa odkazujeme výlučne na typy objektov definované v súbore `objects.xml`, v druhej na akýkoľvek typ definovaný súborom `objects.xml`, alebo súborom špecifickým pre DBMS (napríklad `oracle_objects.xml`). V príklade ukážeme, ako vyzerá základná hierarchia:

```
...
<model-def>
  <hierarchy-forest>
    <hierarchy-tree id="schema_tree">
      <object-ref type="schema">
        <object-ref type="table">
          <object-ref type="table_column"/>
          <object-ref type="primary_key">
            <object-ref type="pk_column"/>
          </object-ref>
          <object-ref type="constraint">
            <object-ref type="con_column"/>
          </object-ref>
        </object-ref>
      </object-ref>
    </hierarchy-tree>
  </hierarchy-forest>
</model-def>
...
```

Príklad: Definícia základnej hierarchie v súbore `generic.machine.xml`

Hierarchia sa definuje v XML sekcii *model-def*. Systém *Schemagic* dovoľuje v hierarchii niekoľko stromov, my budeme používať len jeden, ktorého koreňom je metaobjekt reprezentujúci samotnú schému. Následne v súbore definujeme hierarchiu objektov – koreň tvorí schéma, ktorá môže obsahovať ľubovoľný počet tabuliek. Tabuľka môže obsahovať primárny kľúč, ktorého potomkovia môžu byť stĺpce primárneho kľúča, atď. Konfigurácia neurčuje prípustný počet potomkov daného typu. Napr. nikde nie je definované, že tabuľka môže mať len jeden primárny kľúč. Záväzne však určuje hierarchiu objektov. Definícia hierarchie pre DBMS Oracle v súbore `oracle.machine.xml` vyzerá

podobne, ale odkazuje sa na mená špecifických typov (napr. *ora_table*, *ora_table_column*, atď.).

5.1.2.3 Spôsob získania metadát

V konfiguračných súboroch *Schemagicu* sme zatiaľ nadefinovali typy metaobjektov a vzťahy medzi nimi (hierarchiu metaobjektov). Teraz nás čaká zrejme najdôležitejšia časť a tou je nastavenie spôsobu naplnenia metaobjektov. Potrebujeme zaznamenať pravidlá typu „všetky tabuľky danej schémy v DBMS Oracle sa dajú získať z metadát JDBC drivera“, alebo „všetky stĺpce danej schémy a danej tabuľky v DBMS Oracle sa dajú získať dotazom do špecifikovanej tabuľky”. Ak bude *Schemagic* disponovať takýmito informáciami, dokáže podľa určenej typovej hierarchie vytvoriť XML strom inštancií databázových objektov, ktorý bude reprezentovať databázovú schému.

Je niekoľko spôsobov, ako sa *Schemagic* môže dostať k metainformáciám, tj. k informáciám popisujúcim databázovú schému. Tieto informácie sú uložené v databázovom stroji. Najjednoduchším spôsobom je zrejme použiť metadáta, ktoré poskytuje JDBC driver. Na tieto informácie sa nedá vždy spoľahnúť. Sú závislé na implementátorovi drivera, ktorý nemusí mať úplný prehľad o danom type databáze, alebo o normách, ktoré sa vzťahujú na JDBC drivery. *Schemagic* preto poskytuje ďalšiu možnosť a tou je dotazovanie sa priamo systémových tabuliek. Takéto tabuľky sú prítomné vo väčšine databázových riešení. Napríklad v DBMS Oracle existuje tabuľka `ALL_TABLES`, použitím vhodných podmienok vo WHERE sekcii dotazu môžeme napríklad získať informácie o všetkých tabuľkách vo zvolenej schéme.

Pravidlá na získanie metadát sa zapisujú do rovnakého súboru, v akom je umiestnená hierarchia typov databázových objektov. Pre DBMS Oracle to je teda už spomínaný súbor `oracle.machine.xml`. Opäť uvedieme príklad – informácie o tabuľke v DBMS Oracle môžeme získať dotazom do tabuľky `ALL_TABLES`, tento dotaz teda zapíšeme do konfigurácie:

```
...
<object-load object_type="ora_table" class_name="JdbcSqlDbObjectLoader">
  <parameters>
    <parameter name="SQL_COMMAND" value="select T.TABLE_NAME NAME,
T.NUM_ROWS ROWS_COUNT, T.TABLESPACE_NAME, T.CLUSTER_NAME, C.COMMENTS
USRCOMMENT from ALL_TABLES T, ALL_TAB_COMMENTS C where
T.OWNER=':global[@LOADED_SCHEMA]:' AND C.OWNER(+)=T.OWNER AND
C.TABLE_NAME(+)=T.TABLE_NAME order by T.TABLE_NAME"/>
  </parameters>
  <pre-processing class_name="SQLPreprocessor"/>
</object-load>
...
```

Príklad: Predpis na získanie metadát

Objekt *JdbcSqlDbObjectLoader* dokáže získať dáta z výsledkov dotazu, ktorý mu nastavíme ako parameter SQL_COMMAND. Každý záznam vo výsledku tvorí samostatný metaobjekt a každý stĺpec tohto záznamu tvorí vlastnosť tohto metaobjektu (hodnota stĺpca v zázname tvorí hodnotu tejto vlastnosti). Z toho vyplýva, že dotaz by mal vrátiť stĺpce s názvami, ktoré sa presne zhodujú s definovanými vlastnosťami databázového objektu. Napríklad pre typ *ora_table* by to mali byť stĺpce NAME a ROWS_COUNT zdedené z typu *table* a tiež stĺpce TABLESPACE_NAME, CLUSTER_NAME a USRCOMMENT, ktoré deklaruje priamo typ *ora_table*. Všimnime si, že dotaz v príklade túto podmienku spĺňa a metaobjekt popisujúci existujúci databázový objekt bude všetky tieto vlastnosti obsahovať. V dotaze môžeme používať parametre, napríklad názov aktuálnej schémy, alebo meno rodičovského objektu (napríklad pri následnom získaní všetkých stĺpcov tabuľky budeme potrebovať meno aktuálnej tabuľky). Detaily o nahradzovaní parametrov skutočnými hodnotami, ako aj všetky ostatné detaily tejto konfigurácie sú uvedené v [7].

5.1.2.4 Pripojenie k databáze

Schemagic nám umožňuje získať informácie o akejkoľvek databázovej schéme z databázových strojov, pre ktoré má konfiguračné súbory. Zatiaľ sme však nikde nedefinovali konkrétnu databázu a konkrétnu schému, ktorú má spracovať. Túto informáciu opäť vkladáme do XML súboru, ktorý *Schemagic* dostane ako parameter. Tento súbor teda nemusí mať žiadne špeciálne meno a nemusí byť uložený v podstrome domovského adresára *Schemagicu*. Opäť uvedieme príklad obsahu:

```
...
<use-dbmachine id="oracle"/>
<use-connection id="jdbc"/>
<model-load class_name="DbModelLoader"/>
<parameters>
  <connection id="jdbc">
    <properties>
      <property name="DRIVER" value="oracle.jdbc.driver.OracleDriver"/>
      <property name="URL" value="jdbc:oracle:thin:@or10:1521:XE"/>
      <property name="USER" value="USER1"/>
      <property name="PASSWORD" value="user1"/>
    </properties>
  </connection>
  <object id="global">
    <properties>
      <property name="LOADED_SCHEMA" value="USER1"/>
      <property name="SCHEMA_NAME" value="USER1"/>
    </properties>
  </object>
</parameters>
...
```

Príklad: Informácie o globálnych parametroch a pripojení k DB

V prvom riadku príkladu deklarujeme, že sa budeme pripájať do databázy typu Oracle. *Schemagic* preto použije konfiguračné súbory databázových typov určené pre túto databázu. V druhej časti definujeme parametre – najprv parametre JDBC pripojenia, potom globálne parametre, ktoré sa používajú ako premenné v SQL príkazoch v súbore `oracle.machine.xml`.

Schemagic schému spracuje a výsledok zapíše do určeného XML súboru. Štruktúru výstupného súboru a zároveň vstupného súboru synchronizačnej aplikácie v skratke popíšeme v nasledujúcej sekcii.

5.1.3 Popis schémy v XML

Výstup aplikácie *Schemagic* v XML formáte popisuje určenú databázovú schému. Pozrime sa, ako vyzerá časť takého výstupu:

```
...
<object family="table" type="ora_table">
  <is-type-of type="table"/>
  <is-type-of type="ora_table"/>
  <properties>
    <property key="false" name="CLUSTER_NAME" value=""/>
    <property key="false" name="ROWS_COUNT" value="0"/>
    <property key="true" name="NAME" value="ACTOR"/>
    <property key="false" name="USRCOMMENT" value=""/>
    <property key="false" name="TABLESPACE_NAME" value="USERS"/>
  </properties>
  <contains>
    <object family="table_column" type="ora_table_column">
      <is-type-of type="table_column"/>
      <is-type-of type="ora_table_column"/>
      <properties>
        <property key="true" name="NAME" value="ACTOR_ID"/>
        <property key="false" name="DATA_TYPE" value="NUMBER"/>
        ...
      </properties>
    </object>
    ...
```

Príklad: Časť výstupu aplikácie *Schemagic*

Schemagic postupuje podľa definovaného stromu typov pre daný databázový stroj. Podľa pravidiel na získanie metadát si postupne vyžiada všetky objekty daného typu a k nim ďalej vyhľadáva ich potomkov, až nakoniec zostaví celý strom metaobjektov. Tento strom potom uloží do XML súboru, ktorého časť je uvedená v príklade. V hornej časti sú zachytené informácie o tabuľke ACTOR a jej vlastnosti. Napríklad vieme, že tabuľka je uložená v *tablespace* USERS. Tabuľka má niekoľko stĺpcov, všetky sú zastúpené v XML súbore. V sekcii *contains* tabuľky vidíme definíciu prvého stĺpca ACTOR_ID a jeho vlastnosti, napríklad že tento stĺpec má nastavený dátový typ NUMBER.

Zopakujeme, že vstup do aplikácie na synchronizáciu databázových schém tvoria tri XML súbory. Prvý a druhý získame zo *Schemagicu*, obidva popisujú schému – referenčnú a porovnávanú – a majú práve popísaný formát. Schémy nemusia byť získané z rovnakých databázových strojov. Tretí vstupný súbor je konfiguračný súbor synchronizácie, ktorého základnú štruktúru popíšeme v ďalšej sekcii.

5.1.4 Konfiguračný súbor

Konfiguračný súbor synchronizačnej aplikácie má opäť tvar XML súboru. Obsahuje niekoľko sekcií, každá z nich sa použije v inej fáze procesu porovnávania a synchronizácie. Napríklad časť s definíciami pravidiel o ekvivalencii databázových objektov sa použije v úvodnej časti porovnávania, časť s pravidlami generovania SQL súborov naopak v samotnom závere, pri prechádzaní diff stromu a generovaní výstupu. XML súbor obsahuje aj definície, ktoré sa môžu použiť vo všetkých častiach aplikácie – jedná sa hlavne o parametre a o knižnicu JavaScript funkcií. Pretože sme zatiaľ ešte presne nepopísali tvar pravidiel, alebo presný spôsob integrácie JavaScriptu do aplikácie, v tejto kapitole len naznačíme, čo konkrétne sekcie obsahujú. Detaily popíšeme v častiach, ktoré budú nastavenia v konkrétnych sekciách skutočne využívať.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<options>
    <general>
        <referenceMachine machineType="oracle"/>
        <comparisonMachine machineType="oracle"/>
    </general>
    <parameters>...</parameters>
    <equivalence>...</equivalence>
    <comparison>...</comparison>
    <sqlGen>...</sqlGen>
    <library>...</library>
</options>
```

Príklad: Štruktúra konfiguračného súboru synchronizačnej aplikácie

V konfigurácii používame šesť sekcií. V prvej sekcii (*general*) definujeme, pre aké typy databázových strojov je konfiguračný súbor určený. V tomto prípade sa jedná o porovnanie dvoch schém DBMS Oracle. Je možné, že v budúcnosti do tejto sekcie pribudnú ďalšie globálne nastavenia. V sekcii *parameters* užívateľ definuje parametre, ktoré sa následne môžu používať vo všetkých ďalších častiach. Napríklad sa tu môže definovať parameter určujúci, či sa pri porovnávaní berú do úvahy rozdiely medzi veľkými a malými písmenami, tj. či je porovnávanie case sensitive. Zmena jedného parametra potom ovplyvní správanie celej aplikácie. Nasleduje sekcia *equivalence*, v ktorej budeme uvádzať pravidlá, ktoré jednoznačne určia, na základe akých vlastností sú dva objekty považované za ekvivalentné. Podobne sekcia *comparison* obsahuje pravidlá, podľa ktorých sa postupuje pri porovnávaní ďalších vlastností dvoch (ekvivalentných) objektov. V predposlednej sekcii (*sqlGen*) užívateľ nájde pravidlá týkajúce sa generovania SQL skriptu na základe nájdených zmien. V

poslednej časti, tzv. *library* sekcii, sú definované funkcie napísané v jazyku JavaScript, ktoré sú dostupné z pravidiel všetkých troch predchádzajúcich sekcií. O obsahu jednotlivých sekcií a o spôsobe použitia informácií z konfiguračného XML súboru budú pojednávať ďalšie sekcie.

5.2 Porovnanie schém

Predpokladajme, že užívateľ najprv pomocou aplikácie *Schemagic* vygeneroval popisy dvoch databázových schém a má ich uložené v XML súboroch *reference.xml* a *comparison.xml*. Ďalej predpokladajme, že užívateľ má k dispozícii konfiguračný súbor, ktorý sme zatiaľ len v stručnosti popísali v predchádzajúcej kapitole. Konfiguračný súbor je určený pre zodpovedajúce typy databázových strojov. Sú teda splnené všetky predpoklady na spustenie aplikácie. Presný návod spolu so zoznamom parametrov je uvedený v záverečnej časti práce.

5.2.1 Vyhľadanie ekvivalentov

Vyhľadávanie ekvivalentov sme popísali v kapitole 3.5. Implementácia tohto vyhľadávania vychádza z Algoritmu 1. V praxi však nie je jednoznačné, ako implementovať funkciu *keyPropertiesMatch*. Funkcia má vrátiť *true*, ak majú kľúčové hodnoty databázového objektu rovnaké hodnoty, v opačnom prípade vráti *false*. V prvom rade musíme rozhodnúť, ktoré vlastnosti sú kľúčové – niekedy to nie je úplne zrejmé. Napríklad ak má databázový objekt typu obmedzenie (*constraint*) názov, ktorý bol vygenerovaný systémom, nemalo by sa jednať o kľúčovú vlastnosť. Namiesto názvu by sme mali vybrať inú vlastnosť, alebo vlastnosti.

Ak už rozhodneme, ktoré vlastnosti sú kľúčové, musíme ešte určiť, ako budeme porovnávať hodnoty. Napríklad môžeme porovnávať textovo bez ohľadu na to, či sú v texte použité veľké, alebo malé písmená. Pri niektorých objektoch dokonca nemusíme porovnávať žiadne vlastnosti na to, aby sme rozhodli, či sú navzájom ekvivalentné. Napríklad ak sme už rozhodli o tom, že dve tabuľky sú ekvivalentné, každá má maximálne jeden primárny kľúč; tieto dva kľúče teda musia byť ekvivalentné – to však ešte nič nehovorí o stĺpcoch primárneho kľúča, alebo o jeho ďalších vlastnostiach.

Porovnávanie kľúčových vlastností je teda pomerne variabilné a užívateľovi by sme mali dať možnosť rozhodnúť, ktoré vlastnosti bude porovnávať a akú porovnávaciu metódu zvolí. V niektorých prípadoch bude možno potrebovať aj údaje o iných objektoch, napr. o

potomkoch. Ak má byť aplikácia užitočná, mal by mať užívateľ k dispozícii možnosti aj na takéto zložitejšie porovnania.

Pri vyhľadávaní ekvivalentov, rovnako ako pri porovnávaní ďalších vlastností a generovaní SQL skriptu bude mať užívateľ vždy dve možnosti, ako zadefinovať pravidlo. Prvý spôsob bude jednoduchý, bude riešiť štandardné situácie a predpokladáme, že užívateľ týmto jednoduchým spôsobom vyrieši veľkú väčšinu všetkých prípadov. Tieto pravidlá budeme volať *natívne*. Druhým spôsobom bude použitie skriptovacieho jazyka JavaScript, pomocou ktorého môže užívateľ ľubovoľne pristupovať k uzlom diff stromu, má k dispozícii podmienky, cykly, vlastné užívateľské funkcie a všetky ostatné konštrukty skriptovacieho jazyka. Použitie JavaScriptu vyžaduje isté vedomosti a zručnosti, ale dovoľuje prakticky neobmedzené nastavenie aplikácie. Predvedieme si obidva druhy pravidiel.

5.2.1.1 Natívne pravidlá porovnania

Každé natívne pravidlo musí obsahovať názvy typov databázových objektov, ktoré môžu byť navzájom ekvivalentné, ak spĺňajú ďalšie podmienky. V pravidle uvádzame aj spoločného predka obidvoch objektov. V prípade, že porovnávame objekty rovnakého typu databázového stroja, spoločný predok môže byť samotný typ, tak, ako je to naznačené v príklade. Pravidlo musí obsahovať buď sekciu *native*, alebo sekciu *script*. V tomto prípade chceme definovať natívne pravidlo, preto do sekcie *native* zapíšeme zoznam všetkých kľúčových vlastností objektu *ora_table*:

```
<equivalence>
  ...
  <rule referenceObj="ora_table" comparisonObj="ora_table"
    commonObj="ora_table">
    <native>
      <compare caseSensitive="true" property="NAME"/>
    </native>
  </rule>
  ...
</equivalence>
```

Príklad: Podľa pravidla rozhodneme, či sú dva objekty ekvivalentné

Ak aplikáciu spustíme s konfiguráciou obsahujúcou toto pravidlo, metóda *keyPropertiesMatch* v Algoritme 1 uspeje vtedy a len vtedy, ak budú obidva objekty typu *ora_table* a budú sa zhodovať v hodnote vlastnosti NAME. Inými slovami tabuľky v dvoch rôznych schémach budeme považovať za zodpovedajúce si, ak majú rovnaké meno. Opäť

pripomíname, že to ešte nič nehovorí o zhodnosti hodnôt ostatných vlastností, alebo o tom, akých potomkov obidva objekty majú.

5.2.1.2 Porovnania realizované skriptom

Bez toho, aby sme do detailov rozoberali možnosti, ktoré nám poskytuje použitie jazyka JavaScript (to bude predmetom inej kapitoly), ukážeme si, ako môžeme definovať vyhodnotenie ekvivalentnosti práve kódom tohto skriptovacieho jazyka. Najprv príklad konfigurácie:

```
<equivalence>
  ...
  <rule comparisonObj="ora_table" referenceObj="ora_table"
    commonObj="ora_table">
    <script>
      function ora_table_eq() {
        return referenceObj.NAME == comparisonObj.NAME;
      }
      ora_table_eq();
    </script>
  </rule>
  ...
</equivalence>
```

Príklad: Použitie skriptovacieho jazyka JavaScript

S týmto pravidlom bude aplikácia fungovať rovnako, ako pracovala s natívnym pravidlom, tj. porovná sa hodnota vlastnosti NAME. Pri porovnaní sa však použije JavaScript, konkrétne sa zavolá funkcia *ora_table_eq()*. Výsledok volania skriptu je vždy hodnota posledného výrazu. Keďže posledný výraz v skripte je volanie funkcie, hodnota skriptu bude buď *true* v prípade, že objekty sa zhodujú v názve, alebo *false* inak. Ako volanie skriptu funguje, aké funkcie môžeme používať, ktoré premenné sú globálne, to všetko bude obsahom ďalšej kapitoly.

5.2.2 Používanie JavaScriptu v aplikácii

Integrácia skriptovacieho jazyka JavaScript patrí medzi základné črty systému na synchronizáciu databázových schém. JavaScript môže ľubovoľne pristupovať k uzlom diff stromu a vyhodnocovať zložité výrazy. Jeho použitie je veľmi pohodlné, užívateľ dokonca môže priamo zo skriptov využívať API Javy (tj. vytvárať objekty, volať ich metódy, atď.).

5.2.2.1 Funkcie a premenné prístupné v skriptoch

Skripty sa môžu v našej aplikácii použiť v troch prípadoch – pri rozhodovaní o ekvivalentnosti dvoch objektov, pri porovávaní ďalších vlastností ekvivalentných objektov a pri generovaní výstupného SQL skriptu. Vo všetkých troch prípadoch je najdôležitejší prístup k vlastnostiam spracovávaných objektov, resp. spracovávaného objektu.

Pri porovnaní na ekvivalenciu a pri porovnaní ostatných vlastností potrebuje užívateľ v skripte pristupovať k dvom aktuálnym metaobjektom. Prvý obsahuje vlastnosti databázového objektu v referenčnej schéme, druhý vlastnosti databázového objektu v porovnávanej schéme. Skript môže priamo pristupovať ku globálnym premenným *referenceObj* a *comparisonObj*, ktoré obsahujú aktuálne spracovávané metaobjekty. Ako príklad môžeme použiť porovnanie, ktoré sme videli v predchádzajúcej ukážke kódu: `referenceObj.NAME == comparisonObj.NAME`. Porovnávame vlastnosť dvoch metaobjektov, výsledok je buď hodnota *true*, alebo hodnota *false*. Používame štandardné porovnanie reťazcov vstavané v jazyku JavaScript.

Pri generovaní SQL už nepotrebujeme prístup k dvom objektom, pretože pracujeme už len s jedným diff stromom. Premenná, v ktorej je tento objekt uložený sa volá *currentObj*, k vlastnostiam metaobjektu pristupujeme podobne ako v predchádzajúcom prípade (napr. zápis *currentObj*.NAME vráti názov spracovávaného objektu).

Pre každý metaobjekt, a teda špeciálne aj pre metaobjekty prístupné cez premenné *referenceObj*, *comparisonObj* a *currentObj*, platí, že má zaregistrované tieto ďalšie objektové premenné:

- *parent* – odkaz na predka v strome, napríklad *referenceObj.parent*.NAME vypíše názov rodičovského databázového objektu
- všetky objekty v hierarchii nad daným objektom sú prístupné cez meno ich typu; napr. ak spracúvame objekt typu *ora_table_column*, môžeme k jeho predkom pristupovať takto: *referenceObj.ora_table*.NAME, *referenceObj.ora_schema*.NAME
- *children* – pole potomkov daného objektu, napríklad *referenceObj.children[0]* je prvý potomok aktuálneho objektu

Každý metaobjekt obsahuje metódy, ktoré môže užívateľ zavolať. Sú to:

- *getChildrenByType* – vráti pole potomkov špecifického typu; napríklad ak pracujeme s tabuľkou DBMS Oracle, príkaz

referenceObj.getChildrenByType("ora_table_column") vráti pole metaobjektov, ktoré reprezentujú stĺpce danej tabuľky

- *getPropertyByName* – vráti hodnotu vlastnosti, ktorej názov je v parametri, napríklad *referenceObj.getPropertyByName("TABLESPACE_NAME")*
- *getType* – vráti názov typu metaobjektu, napríklad "ora_table"

V skripte sa dá používať aj globálna funkcia *print*, ktorá sa postará o výpis akejkoľvek hodnoty na štandardný výstup. Napríklad volanie *print(referenceObj.NAME)* vypíše názov aktuálneho databázového objektu na štandardný výstup.

Poslednou globálnou funkciou, ktorú popíšeme v tejto kapitole je funkcia *findObject*. Prvým parametrom je typ objektu, druhým parametrom je hodnota vlastnosti NAME, ktorú má každý databázový objekt. Funkcia vráti inštanciu metaobjektu, alebo null, ak taký metaobjekt neexistuje. Napríklad metaobjekt tabuľky ACTOR môžeme nájsť volaním funkcie *findObject("ora_table", "ACTOR")*.

Existuje ešte niekoľko funkcií a premenných, ktoré sú špecifické pre generovanie SQL skriptu. Tie si priblížime v samostatnej kapitole. V tejto kapitole sme popísali len premenné a funkcie, ktoré môžeme použiť v akomkoľvek prípade. Už podľa množstva týchto pomocných funkcií a premenných je zrejmé, že užívateľ má všetky prostriedky na nastavenie aplikácie tak, aby mu jej výstup plne vyhovoval.

5.2.2.2 Knižnica funkcií

Knižnica spoločných, užívateľom definovaných JavaScript funkcií je ďalším krokom k použiteľnosti celej aplikácie. Existujú totiž úlohy, ktoré treba riešiť často a na mnohých miestach. Užívateľ si môže definovať vlastné funkcie, ktoré potom môže použiť na ľubovoľnom mieste vo svojom JavaScript kóde.

Každá užívateľská funkcia je zaradená do kategórie. Užívateľ sám rozhoduje, aké kategórie bude v konfiguračnom súbore používať. Kategórie sú zavedené len kvôli lepšej orientácii v užívateľských funkciách a nemajú žiadny funkčný význam. Je zaručené, že funkcie z knižnice sú definované pred spustením akéhokoľvek skriptu z pravidla.

Ako príklad použijeme funkciu, ktorá konvertuje postupnosť znakov, označujúcu koniec riadku, na skutočný koniec riadku. Funkcia sa používa často, pretože *Schemagic* automaticky neoddeľuje riadky (napríklad definícií vložených procedúr, alebo pohľadov).

```
<library>
  <function name="replaceEolns" category="util">
    function replaceEolns(textWithEolns) {
      var oldStr = java.lang.String(textWithEolns);
      var newStr = oldStr.replaceAll(&quot;--EOLN--&quot;,
&quot;\n&quot;);
      return newStr;
    }
  </function>
  ...
</library>
```

Príklad: Definícia knižničnej funkcie v konfiguračnom súbore

V príklade si môžeme všimnúť používanie špeciálnej sekvencie znakov *&quot*;. Táto sekvencia je tu kvôli tomu, že znak úvodzoviek je v XML špeciálny znak a nemôžeme ho používať – ak takýto znak chceme uložiť do XML súboru, musíme použiť práve špeciálnu sekvenciu znakov, ktorá sa začína znakom *&*.

Okrem tejto "anomálie" vidíme, že definícia funkcie je štandardná. Jej sekcia obsahuje atribúty určujúce názov funkcie a názov kategórie, do ktorej funkcia patrí. Potom nasleduje hlavička funkcie a jej telo. Od tejto chvíle môžeme funkciu *replaceEolns* používať na ostatných miestach.

5.2.2.3 Používanie API Java 2 SE

Vo funkcii *replaceEolns* sme použili metódu objektu *String* zo štandardnej knižnice jazyka Java. Konkrétne išlo o metódu *replaceAll*, ktorá nahradí regulárny výraz za postupnosť znakov; viac informácií o tejto metóde čitateľ nájde v [19]. To nie je jediný prípad volania metódy jazyka Java zo skriptovacieho jazyka.

Na integráciu JavaScriptu do Javy používame open source knižnicu *Rhino*, ktorá je súčasťou projektu *mozilla.org*. *Rhino* umožňuje používanie API Java 2 Standard Edition priamo zo skriptov napísaných v JavaScripte, aj keď toto prepojenie nie je podporované priamo štandardom ECMA [20]. V rovnakom dokumente [20] je podrobný popis možností využitia tohto prepojenia.

5.2.2.4 Výsledok volania skriptu

Ak v pravidle, v ktorejkoľvek z troch sekcií, kde je dovolené používať skripty, použijeme natívne pravidlo, typ jeho výsledku je dopredu definovaný. Pri rozhodovaní

o ekvivalencii dvoch objektov je to *boolean* hodnota. Podobne pri porovnávaní ďalších vlastností ekvivalentných objektov sa pri použití natívneho pravidla rozhodne o tom, či sú objekty zhodné, alebo nie. Tj. výsledok je opäť typu *boolean*. Navyše sa zaznamená, ktoré vlastnosti neboli zhodné. V poslednej sekcii – pri generovaní výsledného SQL skriptu – natívne pravidlo vracia textový reťazec – SQL príkaz. Navyše môže pre každú skupinu vrátiť ďalší textový reťazec, ktorý bude v skripte umiestnený na odlišnom mieste, tak ako sme to popísali v kapitole 3.9. Výsledkom teda je zoznam dvojíc v tvare *<SQL kód, skupina>*.

Pretože skriptovací jazyk len dopĺňa natívne pravidlá, je zrejmé, že výsledok volania skriptu by mal byť typovo rovnaký, ako výsledok natívneho spôsobu.

Platí, že výsledná hodnota skriptu vloženého do pravidla je hodnota posledného výrazu v skripte. Nepoužíva sa žiadna *return* hodnota, pretože skript nie je „obalený“ funkciou. Najjednoduchší a odporúčaný spôsob, ako správne vrátiť hodnotu požadovaného typu je vytvoriť si v pravidle vlastnú funkciu a tú potom zavolať vhodným volaním, tak, ako sme to videli v príklade. Volanie funkcie je teda posledný príkaz kódu a teda výsledok skriptu je čokoľvek, čo v sekcii *return* vráti vytvorená funkcia.

Keďže minimálne v dvoch z troch prípadov by sme potrebovali vrátiť zložitejšiu dátovú štruktúru, mohli by sme takú štruktúru v jazyku JavaScript vytvoriť, vo funkcii naplniť a vrátiť príkazom *return*. Tento prístup sa nám nezdal dostatočne flexibilný a tak sme pristúpili k alternatívnemu riešeniu. Z funkcie vždy vraciame hodnoty jednoduchého typu, prípadne žiadnu hodnotu. Na uloženie zložitejších dát používame vopred pripravené globálne JavaScript funkcie so zodpovedajúcimi parametrami.

5.2.3 Porovnanie ďalších vlastností ekvivalentných objektov

V kapitole 3.6 sme popísali, čo sa stane, keď sa dva databázové objekty označia ako ekvivalentné (zodpovedajúce si). Tieto objekty sa určite zhodujú v kľúčových vlastnostiach, treba však porovnať aj ostatné vlastnosti, aby sa zistilo, či sú zhodné. Niektoré vlastnosti porovnávať nechceme, pretože pre nás nie sú podstatné – napríklad pozícia stĺpca v tabuľke. Ak stĺpec s rovnakým názvom a s rovnakým dátovým typom existuje v obidvoch schémach, na jeho fyzickej pozícii v tabuľke nám nezáleží (premiestnenie stĺpca na inú pozíciu by sme aj tak vedeli dosiahnuť len vymazaním a opätovným vytvorením tabuľky).

Zoznam porovnávaných vlastností pre každý typ objektu opäť definujeme pravidlami v XML konfiguračnom súbore. Pravidlá sa podobajú na pravidlá z kapitoly 5.2.1. Môžu byť buď natívne, alebo môžeme na porovnanie použiť skriptovací jazyk.

Opäť uvedieme príklad základného natívneho pravidla. V tomto pravidle vymenujeme vlastnosti databázového objektu typu tabuľka DBMS Oracle, ktoré sa majú porovnávať:

```
<comparison>
  <rule commonObj="ora_table">
    <native>
      <compare caseSensitive="true" property="TABLESPACE_NAME"/>
      <compare caseSensitive="true" property="CLUSTER_NAME"/>
      <compare caseSensitive="true" property="USRCOMMENT"/>
      <compare caseSensitive="true" property="NAME"/>
    </native>
  </rule>
  ...
</comparison>
```

Príklad: Natívne pravidlo porovnávania vlastností

V hlavičke týchto pravidiel už neuvádzame typ referenčného objektu, typ porovnávaného objektu a typ spoločného predka, ako tomu bolo v pravidlách ekvivalencie, ale stačí nám informácia o spoločnom predkovi. Tento typ sa musí zhodovať s typom, ktorý sme uviedli do zodpovedajúceho pravidla ekvivalencie (*commonObj*).

Zoznam vlastností môže obsahovať len vlastnosti prítomné v type *commonObj*. Pre každú vlastnosť samostatne špecifikujeme, či budeme rozlišovať veľké a malé písmená, alebo nie. Výsledkom porovnania je hodnota typu boolean, ktorá nás informuje o tom, či sú objekty zhodné, alebo sa v hodnotách niektorých vlastností líšia. Pri spracovaní výstupu nás však nebude zaujímať len to, či sú objekty rovnaké, alebo rôzne, budeme potrebovať aj zoznam rozdielnych vlastností. Pri natívnom porovnaní sa o to postará samotná aplikácia a pre každý databázový objekt si tieto odlišné vlastnosti zaznamená.

Pri niektorých databázových objektoch nám opäť nebude stačiť natívne porovnanie, pretože budeme potrebovať použiť špeciálne metódy porovnania, alebo niektoré vlastnosti porovnávať len vtedy, ak sú splnené isté podmienky. V týchto prípadoch opäť využijeme integrovaný skriptovací jazyk. V zápise pravidla nahradíme sekciu *native* za sekciu *script* a do nej vložíme porovnávaciu logiku.

Výsledok vykonania skriptu bude opäť hodnota typu *boolean*, ktorá informuje o tom, či sú hodnoty všetkých porovnávaných vlastností rovnaké, alebo nie. Ak zistíme, že sú niektoré vlastnosti rôzne, musíme to sami zaznamenať, pretože aplikácia sa o to v tomto prípade nemôže starať automaticky. Na to nám bude slúžiť globálna premenná *differentProperties*. V prípade, že sa hodnoty vlastnosti v referenčnom a v porovnávanom

objekte nezhodujú, vložíme názov vlastnosti do globálnej premennej takto: *differentProperties.add(propName)*.

V preddefinovaných konfiguráciách existujú v knižnici skriptovacích funkcií funkcie *compareProperty* a *compareProperties*, ktoré porovnajú zvolené vlastnosti a dokážu automaticky zapísať názvy zmenených vlastností do premennej *differentProperties*. Funkcia na porovnanie vlastností databázového objektu typu *ora_constraint* potom môže vyzerať napríklad takto:

```
function ora_constraint_comp() {
  var flag = true;
  if (referenceObj.GENERATED != "GENERATED NAME") {
    flag = flag && compareProperty("NAME", true, true);
  }
  flag = flag && compareProperties(new Array("DEFERRABLE", "VALIDATED",
"RELY", "DEFERRED", "STATUS", "CHECK_CONDITION"), true, true);
  return flag;
}

ora_constraint_comp();
```

Príklad: Funkcia porovnáva vlastnosti dvoch obmedzení (*constraint*)

Príklad ukazuje podmienečné porovnanie vlastnosti NAME. Ak by sme vytvorili natívne pravidlo, nevedeli by sme v ňom zabezpečiť rovnakú logiku, ako v uvedenej funkcii. V prípade, že obmedzenie (*constraint*) v referenčnej schéme má systémom vygenerované meno, nebudeme túto vlastnosť vôbec porovnávať. Je pre nás nepodstatná, pretože vygenerovaný názov v jednej databázovej schéme sa takmer určite líši od vygenerovaného názvu v druhej databázovej schéme, pričom názov pri tomto druhu databázového objektu nie je podstatný. Vo funkcii voláme knižničné funkcie *compareProperty* a *compareProperties*, ktoré sa automaticky starajú o prípadné zaznamenanie názvov vlastností, ktorých hodnoty sa líšia.

5.3 Výstup aplikácie

Porovnanie databázových objektov a vytvorenie tzv. diff stromu podľa Algoritmu 2 tvorí len jednu časť synchronizačnej aplikácie. Druhou, nemenej dôležitou časťou je generovanie výstupu. Kým textový výstup a výstup vo forme XML súboru vytvoríme relatívne jednoducho jednoduchou transformáciou diff stromu, pri generovaní synchronizačného SQL skriptu budeme opäť musieť zaviesť užívateľské pravidlá s podporou skriptovacieho jazyka, ktoré zabezpečia funkcionalitu naznačenú v analýze.

5.3.1 Základné výstupné súbory

Za základné výstupné súbory považujeme textový výstup pre užívateľa a výstup vo formáte XML, ktorý je určený na ďalšie strojové spracovanie. Existuje ešte jeden druh výstupu a tým je grafické znázornenie diff stromu v GUI. Tento výstup podrobnejšie popíšeme v kapitole 6.5.5.

Všetky tri výstupy vzniknú prechádzaním diff stromu podľa Algoritmu 3 popísaného v analytickej časti. Tieto výstupy nepotrebujeme nijako konfigurovať, v konfiguračnom súbore preto pre ne neexistuje žiadna špeciálna sekcia.

5.3.2 Generovanie SQL

Poslednou formou výstupu je SQL skript, ktorý po spustení v databáze porovnávanej schémy túto schému upraví tak, aby z hľadiska svojej štruktúry zodpovedala referenčnej schéme. Analýzu tejto časti sme uviedli v kapitole 3.9, v tejto kapitole sa budeme venovať jej implementácii.

5.3.2.1 Skupiny SQL príkazov

V sekcii *sqlGen* konfiguračného súboru najprv určíme poradie skupín SQL príkazov. Každý príkaz môžeme zaradiť do jednej konkrétnej skupiny. Ak skupinu nedefinujeme, automaticky sa zaradí do skupiny *[default]*. Príklad zoradenia skupín:

```
<sqlGen>
  <groups>
    <group>dropTriggers</group>
    <group>dropConstraints</group>
    <group>[default]</group>
    <group>addFKs</group>
    <group>addTriggers</group>
    <group>addViews</group>
    <group>addSynonyms</group>
  </groups>
  ...
</sqlGen>
```

Príklad: Zoradenie skupín

V každej skupine sa SQL príkazy zoraďujú v takom poradí, v akom ich do nej postupne zaraďujeme. Výsledný skript bude najprv obsahovať SQL príkazy z prvej skupiny (*dropTriggers*), po nich budú nasledovať príkazy z druhej skupiny (*dropConstraints*), atď. Ak niektorú skupinu neuvedieme do tejto konfigurácie a napriek tomu do nej zaraďujeme

SQL príkazy, nenastane chyba. Príkazy sa však vo výslednom SQL skripte jednoducho nebudú vyskytovať.

5.3.2.2 Pravidlá generovania SQL výstupu

Po definícii poradia skupín nasledujú pravidlá, podľa ktorých generujeme výstup. Opäť platí, že existujú dva prístupy – natívne pravidlá a použitie skriptu.

Pravidlá sa vždy definujú pre istý typ objektu. Napríklad pre databázový objekt tabuľka. Platí, že v diff strome už majú všetky metaobjekty taký typ, ako bol určený spoločný typ v pravidle pre ekvivalenciu, tj. typ uvedený v *commonObj* zodpovedajúceho ekvivalenčného pravidla. Ak napríklad porovnávame schému DBMS Oracle so schémou DBMS MySql, tabuľka v prvej schéme má vždy typ *ora_table* a tabuľka v druhej schéme typ *my_table*. Keďže v pravidle ekvivalencie je určené, že spoločný typ je typ *table*, v diff strome už akákoľvek tabuľka, označená ako nezmenená, nová, alebo zastaralá, bude mať typ *table*. Pravidlá na generovanie SQL výstupu vytvárame už pre tento spoločný typ, nie pre pôvodný typ. Samozrejme, ak porovnávame dve schémy DBMS Oracle, budeme používať typ *ora_table*, pretože ten bol uvedený ako *commonType* v pravidle o ekvivalencii.

Pre každý typ objektu existujú tri nezávislé druhy pravidiel. Delia sa podľa toho, aký je stav databázového objektu – či sa jedná o nový (*new*) objekt, o zastaralý (*obsolete*) objekt, alebo o objekt so zmenenými vlastnosťami (*changed*). Pre každý objekt a pre každú z týchto kategórii môže existovať niekoľko pravidiel.

Každé pravidlo, či už sa jedná o pravidlo s natívnym obsahom, alebo pravidlo, ktorého logika je realizovaná skriptom, môže definovať vstupné podmienky. Pravidlo je realizované len vtedy, ak sú podmienky splnené. Platí, že sa vykoná vždy prvé pravidlo, ktorého podmienky sú splnené – ďalšie pravidlá v poradí sa už nevykonávajú. Preto pravidlá zoraďujeme od špecifických, tj. takých, ktoré majú reštriktívne podmienky, ku všeobecným.

Pravidlá sa do konfiguračného súboru zapisujú ku konkrétnemu typu objektu do troch kategórii (*newRules*, *obsoleteRules*, *changedRules*), základná štruktúra je znázornená v nasledujúcom príklade:

```
<sqlGen>
   ...
   <object name="ora_table">
        <newRules>...</newRules>
        <obsoleteRules>...</obsoleteRules>
        <changedRules>...</changedRules>
   </object>
   ...
</sqlGen>
```

Príklad: Pravidlá generovania pre typ *ora_table*

Formát pravidiel v jednotlivých skupinách je rovnaký. Pravidlo má definované meno a nepovinnú podmienku. Nasleduje sekcia s natívnym obsahom (*native*), alebo skript jazyka JavaScript (*script*).

```
...
<object name="ora_table">
   <newRules>
      <rule name="table-create">
         <condition>...</condition>
         <native>...</native>
      </rule>
      ...
   </newRules>
   ...
</object>
...
```

Príklad: Natívne pravidlo *table-create*

V nasledujúcich kapitolách sa budeme postupne venovať rôznym typom podmienok a spôsobu ich vyhodnocovania, natívnemu obsahu pravidiel a nakoniec pravidlám, ktoré využívajú skriptovací jazyk.

5.3.2.3 Vyhodnocovanie podmienok

Podmienky sa používajú na rozhodnutie, či sa má vykonať pravidlo, resp. úloha v pravidle (viď sekcia 5.3.2.4). Keďže generovanie výsledného SQL skriptu by sa bez podmienok neobišlo, v prípade, že by sme ich do systému nezabudovali, stali by sa natívne pravidlá v podstate nepoužiteľné a užívateľ by bol odkázaný len na konfiguráciu prostredníctvom kódu v jazyku JavaScript.

Existujú tri druhy podmienok. Prvý druh testuje stav konkrétneho databázového objektu, napríklad či je tabuľka, ktorá vlastní práve spracovávaný stĺpec, nová (*new*). Druhý typ podmienky je určený na testovanie hodnoty vybranej vlastnosti, prípadne na jej porovnanie s hodnotou konkrétnej vlastnosti iného databázového objektu. Posledný typ umožňuje testovať, či nejaká vlastnosť patrí do množiny zmenených vlastností. Tento typ podmienky má význam len pri databázových objektoch, ktoré majú niektoré vlastnosti zmenené (stav databázového objektu je *changed*).

Podmienky sa nemusia vzťahovať len k aktuálne spracovávanému objektu. Testovať môžeme aj všetkých predkov objektu. Adresovať ich môžeme menom typu – napríklad v pravidle vzťahujúcemu sa k typu *ora_table* môžeme napísať podmienku na všetkých predkov – v tomto prípade je to len databázový objekt *ora_schema*. Tj. v pravidle môžeme porovnávať databázový objekt *ora_table* a predka *ora_schema*. V prípadnom pravidle pre stĺpec tabuľky môžeme vytvárať podmienky týkajúce sa tohto objektu a jeho predkov; na identifikáciu použijeme názvy typov: *ora_table_column*, *ora_table* a *ora_schema*.

Každá podmienka pozostáva z nula až neobmedzene veľa výrazov (budeme ich nazývať aj *items*, napríklad výraz na testovanie stavu objektu bude v XML označený značkou *stateItem*). Výsledok podmienky sa určí aplikovaním logického AND na výsledky jednotlivých výrazov. Inak povedané – celá podmienka bude považovaná za splnenú len vtedy, ak sú splnené všetky jej časti (výrazy).

5.3.2.3.1 Testovanie stavu databázového objektu

Zrejme nemá veľký význam testovať stav databázového objektu, pre ktorý píšeme pravidlo, keďže stav je určený kategóriou, v ktorej je pravidlo umiestnené. Ak napr. pravidlo umiestnime do sekcie *newRules*, bude sa vyhodnocovať len vtedy, ak je aktuálny objekt považovaný za nový (*new*). Väčšinou teda testujeme stavy predkov. Ak sa napríklad v diff strome nachádza stĺpec tabuľky, ktorý je označený ako nový (*new*), zrejme by sme mali vygenerovať vhodný príkaz typu ALTER TABLE, ktorý pridá stĺpec do tabuľky. To je správny prístup, až na jednu výnimku – ak je totiž *nová* aj nadradená tabuľka, nemusíme robiť nič, pretože všetky nové stĺpce tejto novej tabuľky budú vytvorené v pravidle tabuľky príkazom CREATE TABLE. To znamená, že na nový stĺpec chceme reagovať len vtedy, ak nadradená tabuľka nie je nová. Túto podmienku zapíšeme nasledovne:

```
...
<condition>
  <stateItem objectName="ora_table" operator="isNot">
    <state>new</state>
  </stateItem>
</condition>
...
```

Príklad: Testovanie stavu objektu *ora_table*

Zoznam operátorov, ktoré môžeme použiť v teste na stav databázového objektu je uvedený v závere práce.

5.3.2.3.2 Testovanie hodnoty vlastnosti databázového objektu

Podmienka môže obsahovať test na hodnotu vlastnosti konkrétneho objektu. Napríklad môžeme testovať, či je vlastnosť CONSTRAINT_TYPE objektu *ora_constraint* nastavená na hodnotu "R". To by znamenalo, že sa jedná o obmedzenie popisujúce cudzí kľúč (*foreign key*) a na takéto obmedzenie môžeme reagovať inak, ako na ostatné typy obmedzení.

```
...
<condition>
  <propertyItem objectName="ora_constraint" property="CONSTRAINT_TYPE"
operator="is" comparisonValue="R" />
</condition>
...
```

Príklad: Porovnávanie vlastnosti CONSTRAINT_TYPE objektu typu *ora_constraint*

Tento typ testu je variabilný – užívateľ môže testovať, či je hodnota vlastnosti prázdna (*null*), alebo môže porovnať vlastnosť jedného objektu s vlastnosťou druhého objektu. Viac informácii nájde čitateľ v závere práce.

5.3.2.3.3 Testovanie zmenených vlastností

Testovanie zmenených vlastností má zmysel len pri objektoch v stave zmenený (*changed*). Napríklad nás môže zaujímať, či bola zmenená aspoň jedna z vlastností, ktoré súvisia s dátovým typom stĺpca tabuľky. Ak áno, môžeme na zmenu reagovať príkazom ALTER TABLE a zo všetkých relevantných vlastností vygenerovať (napr. JavaScript funkciou) nový dátový typ. Uvádzame príklad definície spomínaného testu:

```
...
<condition>
  <changedPropertiesItem objectName="ora_table_column"
operator="listContainsAtLeastOne" >
     <property>DATA_PRECISION</property>
     <property>DATA_SCALE</property>
     <property>DATA_LENGTH</property>
     <property>DATA_TYPE</property>
     <property>IS_NULLABLE</property>
     <property>DEFAULT_VALUE</property>
  </changedPropertiesItem>
</condition>
...
```

Príklad: Test na prítomnosť uvedených vlastností v zozname zmenených vlastností

Test vráti *true* vtedy a len vtedy, ak je v zozname zmenených vlastností aspoň jedna z vymenovaných vlastností. To je presne to, čo od tohto testu očakávame – ak sa totiž zmenila jedna z vymenovaných vlastností, je zrejmé, že sa musí zmeniť aj definovaný dátový typ. Opäť pripomíname, že zoznam dostupných operátorov je závere práce.

5.3.2.4 Natívne pravidlá

Pri návrhu pravidiel sme opäť postupovali tak, aby užívateľ mohol väčšinu potrebných pravidiel zrealizovať natívne a nepotreboval na to skriptovací jazyk. Aby boli natívne pravidlá dostatočne flexibilné, musia byť splnené nasledovné podmienky:

- možnosť nahradzovania premenných hodnotou
- možnosť volania knižničných JavaScript funkcií
- definovanie sekvencie tzv. úloh (*tasks*), ktorých výstup tvorí výsledný SQL skript
- vstupné podmienky úlohy, ktoré rozhodnú, či sa úloha má vykonať
- možnosť vytvoriť cyklus cez všetkých potomkov istého typ

Natívne pravidlá obsahujú tzv. úlohy, menšie celky, ktoré sa priamo vykonávajú. Úloha väčšinou obsahuje základnú kostru výstupného SQL textu, vhodne doplnenú premennými, ktoré sa pri vykonávaní pravidla nahradia skutočnou hodnotou. Výstup úloh sa zaradí buď do výstupnej skupiny *[default]*, ak úlohu definujeme v sekcii *sequence*, alebo do špecifikovanej výstupnej skupiny, ak úlohu definujeme v sekcii *preliminary*. Úlohy sa vykonávajú za sebou, tak, ako sú definované. To má význam hlavne v časti *sequence*, pretože užívateľ môže výsledný text vrátiť po častiach, ktoré rozdelí medzi jednotlivé úlohy. Ukážeme si príklad natívneho pravidla:

```
...
<rule name="function-create">
   <condition/>
   <native>
      <preliminary>
         <simpleTask group="dropFunctions" currentObjectType="ora_function"
name="dropOldFunction">
            ...
         </simpleTask>
      </preliminary>
      <sequence>
         <simpleTask currentObjectType="ora_function" name="addFunction">
            ...
         </simpleTask>
      </sequence>
   </native>
</rule>
...
```

Príklad: Definovanie úloh (tasks)

Výstup z úlohy s názvom *dropOldFunction* sa v celkovom výstupe objaví spolu s ostatnými príkazmi zo skupiny *dropFunctions*, pretože táto úloha je zaradená v sekcii *preliminary* a ako skupinu má uvedenú práve skupinu *dropFunctions*. Výstup úlohy *addFunction* sa zobrazí v rámci štandardnej skupiny *[default]*.

Poznáme úlohy dvoch typov. Prvý typ je už spomenutý *simpleTask* – hlavnou činnosťou tohto základného typu je vygenerovať časť SQL skriptu, ktorý obslúži udalosť, pre ktorú píšeme pravidlo. Druhým typom je tzv. *loopTask* – cyklus, ktorý sekvenčne prechádza potomkov aktuálneho objektu konkrétneho typu a pre každý z nich spustí ďalšiu sekvenciu úloh, ktorá môže ďalej obsahovať buď jednoduché úlohy (*simpleTasks*), alebo ďalšie cykly (*loopTasks*).

5.3.2.4.1 SimpleTask

Každá úloha typu *SimpleTask* má svoje meno, skupinu vykonávania (ak je zaradená do sekcie *preliminary*) a do XML konfigurácie sa uvádza aj typ aktuálne spracovávaného databázového objektu. Môže sa zdať, že tento údaj je redundantný, pri cykloch však zistíme, že výrazne uľahčuje orientáciu v konfiguračnom súbore. Telo úlohy môže obsahovať vstupnú podmienku (rovnakého tvaru, ako vstupná podmienka pravidla). Poslednou časťou je text SQL príkazu, ktorý môže obsahovať premenné, prípadne volania knižničnej funkcie.

Premenná sa v texte vymedzuje špeciálnou sériou znakov takto: *${nazov_premennej.VLASTNOST}*, napríklad *${ora_table.NAME}*. Podobne, ako pri podmienke platí, že používame názov typu objektu na adresovanie samotného objektu. Identifikátor *ora_table* označuje typ, ale my tento názov typu používame ako odkaz na konkrétnu inštanciu. K dispozícii máme – opäť zhodne s tvorbou podmienky – inštancie všetkých predkov aktuálneho objektu, tj. ak píšeme úlohu pre typ *ora_table*, máme k dispozícii aj premennú *ora_schema*.

Existujú aj situácie, keď potrebujeme vlastnosti objektu spracovať nejakou funkciou. Napríklad dátový typ novovytvoreného stĺpca sa nedá určiť len z jednej vlastnosti metaobjektu. Funkcia by mala zobrať do úvahy všetky relevantné vlastnosti (sú to tieto: DATA_PRECISION, DATA_SCALE, DATA_LENGTH, DATA_TYPE, IS_NULLABLE, DEFAULT_VALUE) a podľa ich konkrétnych hodnôt vygenerovať popis dátového typu stĺpca. Použitím znakov *${[funkcia(parametre);]}* zavoláme ľubovoľnú funkciu, ktorá existuje v knižnici (viď sekcia 5.2.2.2). Výsledok funkcie by mal byť textový reťazec, ktorým sa automaticky nahradí toto volanie funkcie. Je dôležité si uvedomiť, že používame premenné a funkcie skriptovacieho jazyka. Napríklad premenná *currentObj* ukazuje na aktuálne spracovávaný objekt diff stromu. Ekvivalentne môžeme použiť aj názov typu spracovávaného objektu, napr. *ora_table_column*. Podrobný popis premenných a funkcií nájde čitateľ v kapitole 5.2.2.1. Pridanie stĺpca do existujúcej tabuľky ukážeme v nasledujúcom príklade:

```
...
<sequence>
  <simpleTask currentObjectType="ora_table_column" name="addColumn">
    <condition/>
    <text>
ALTER TABLE ${ora_table.NAME}
  ADD COLUMN (${ora_table_column.NAME} ${[ getDataType(currentObj); ]}
    ${[ getNotNull(currentObj); ]} ${[ getDefault(currentObj); ]});
    </text>
  </simpleTask>
</sequence>
...
```

Príklad: Pridanie nového stĺpca do už existujúcej tabuľky

Výsledok úlohy je teda text s nahradenými všetkými premennými a volaniami funkcií. Premenná `${ora_table.NAME}` sa nahradí názvom rodičovskej tabuľky, premenná `${ora_table_column.NAME}` sa nahradí názvom novovytvoreného stĺpca. Ostatné výrazy

sú volania funkcií – každá z funkcií je zodpovedná za časť výsledného kódu. Funkcia *getDataType* vytvorí textový reťazec s predpisom dátového typu, *getNotNull* vráti reťazec *NOT NULL* v prípade, ak je príslušná vlastnosť nastavená na *true*, *getDefault* vráti deklaráciu štandardnej hodnoty, ak má pre stĺpec existovať. Vo všetkých prípadoch používame ako parameter premennú jazyka JavaScript *currentObj*, ktorá je naviazaná na aktuálne spracovávaný metaobjekt (v tomto prípade to je stĺpec tabuľky).

5.3.2.4.2 LoopTask

Aby boli natívne pravidlá dostatočne flexibilné, zahrnuli sme do nich možnosť cyklov cez potomkov databázových objektov. Typický príklad využitia sme už spomenuli – pri vytvorení novej tabuľky môžeme pomocou cyklu cez stĺpce tejto tabuľky vytvoriť komplexný príkaz CREATE TABLE *názov_tabuľky* (*zoznam_stĺpcov*). Nová tabuľka by sa v niektorých DBMS možno dala vytvoriť aj bez toho – pri spracovaní objektu tabuľky by sa vytvorila prázdna tabuľka bez stĺpcov a pri spracovaní jednotlivých stĺpcov by sa do tabuľky pridali príkazom ALTER TABLE. Takémuto riešeniu sa však pokúsime vyhnúť.

Cyklus definujeme podobne ako klasickú úlohu. Môžeme ho vložiť buď do časti *preliminary*, alebo do časti *sequence*. Povoľujeme aj vnorené cykly, tj. cyklus, vložený do iného cyklu. Existuje niekoľko parametrov, ktoré musíme určiť:

- typ rodičovského objektu (ten je vždy určený, údaj v definícii len opakujeme)
- typ iterácie
- text, ktorý bude predchádzať vygenerovanému textu z cyklu
- text, ktorý sa vloží za vygenerovaný text z cyklu
- text, ktorý vložíme medzi dve iterácie
- úlohy, ktoré sa budú vykonávať pre každú iteráciu

Typ rodičovského objektu je určený nadradeným objektom. Typ iterácie je typ, pre ktorý celý cyklus realizujeme. V uvedenom príklade je typ rodičovského objektu *ora_table*, typ iterácie je *ora_table_column*. Rodičovský objekt nemusí vždy byť práve spracovávaný databázový objekt, pre ktorý píšeme pravidlo. Pre vnorený cyklus je nadradený typ vždy rovnaký, ako typ iterácie nadradeného cyklu.

Texty, ktoré sa vkladajú pred, za a medzi jednotlivé iterácie zjednodušujú užívateľovi prácu. Podobná funkcionalita by sa dala dosiahnuť napríklad tak, že by sa tieto oddeľovače vkladali do textu v rámci podriadených úloh, to by však nebolo prehľadné. Užívateľ môže rozhodnúť, či sa pred/za text vygenerovaný cyklom vložia oddeľovače aj vtedy, ak je tento

text prázdny, tj. ak by napríklad tabuľka neobsahovala žiadne stĺpce, čo je síce krajne nepravdepodobné, ale podobná situácia môže nastať pri iných objektoch.

Telo cyklu obsahuje opäť buď jednoduché úlohy, alebo ďalšie cykly. Keďže samotný cyklus negeneruje žiadny text, okrem oddeľovačov, táto povinnosť zostáva úlohám v tele cyklu. Definícia úlohy, ani jej použitie sa v podstate nijako nelíši od úlohy, ktorá je zaradená priamo v sekcii *sequence*.

Ako príklad uvedieme už popísanú situáciu, kedy potrebujeme v porovnávanej schéme vytvoriť tabuľku, ktorá existuje v referenčnej schéme, ale neexistuje v porovnávanej schéme:

```
...
<sequence>
  <simpleTask name="createTable" currentObjectType="ora_table">
    <condition/>
    <text>CREATE TABLE ${ora_table.NAME}</text>
  </simpleTask>
  <loopTask parentObjectType="ora_table"
loopObjectType="ora_table_column">
    <insertBefore putIfZeroIterations="true"> ( </insertBefore>
    <insertBetween>, </insertBetween>
    <appendAfter putIfZeroIterations="true">);</appendAfter>
    <tasks>
      <simpleTask name="createColumn"
currentObjectType="ora_table_column">
        <condition/>
        <text> ${ora_table_column.NAME} ${[ getDataType(currentObj); ]}
${[ getNotNull(currentObj); ]}</text>
      </simpleTask>
    </tasks>
  </loopTask>
</sequence>
...
```

Príklad: Pravidlo pre vytvorenie novej tabuľky

V prvom kroku v rámci štandardnej úlohy (*simpleTask*) vytvoríme prvú časť príkazu CREATE TABLE s názvom tabuľky. V druhom kroku iterujeme cez všetky stĺpce zodpovedajúceho metaobjektu, na začiatku vložíme do textu sériu znakov ” (” a to bez ohľadu na to, či iterácia obsahuje aspoň jeden stĺpec. Pri každej iterácii vykonáme vnorenú úlohu, ktorá do textu vloží názov stĺpca a definíciu jeho typu. Medzi tieto definície sa automaticky vloží čiarka, ktorú užívateľ definoval ako oddeľovací znak v sekcii *insertBetween*. Po spracovaní všetkých stĺpcov sa do textu vloží reťazec ”); ”.

5.3.2.5 Pravidlá vykonávané skriptom

Napriek tomu, že natívne pravidlá sú relatívne silné, občas sa môže stať, že budeme potrebovať ešte väčšiu flexibilitu. Namiesto natívneho obsahu do pravidla môžeme vložiť skript jazyka JavaScript. Stačí nahradiť sekciu *native* za sekciu *script* a do nej priamo vložiť kód.

V skripte môžeme pristupovať k diff stromu, máme k dispozícii rôzne funkcie a premenné, ktoré nám to umožňujú, presne podľa kapitoly 5.2.2.1. Zo získaných informácii skript vytvorí textový reťazec, ktorý chce zaradiť do SQL výstupu. Zostáva špecifikovať, akým spôsobom môže užívateľ definovať výstup.

Keďže natívna časť produkuje dvojice typu *<sql text, prioritná skupina>*, pričom skupina určuje, kde vo výstupe sa daný text bude nachádzať, nebude nám stačiť vrátenie jednoduchého textového reťazca. Na poslednej hodnote skriptu nám teda tento krát nebude záležať, všetky uloženia častí textu do finálneho výstupu budeme musieť zrealizovať sami použitím vstavanej funkcie *addSqlText*. Prvý parameter tejto funkcie je samotný text, prípadný druhý parameter určuje skupinu, v ktorej sa má text objaviť. Ak druhý argument neuvedieme, alebo ho definujeme a jeho hodnota bude reťazec *"[default]"*, text sa zobrazí v hlavnej časti SQL skriptu. Ukážeme si príklad funkcie, ktorá vygeneruje príkazy na odstránenie zastaralých (*obsolete*) obmedzení (*constraints*) z porovnávanej databázovej schémy:

```
function dropConstraintSql() {
  var s = "ALTER TABLE " + currentObj.parent.NAME +
       " DROP CONSTRAINT " + currentObj.NAME +"\n\n";
  addSqlText(s, "dropConstraints");
}

dropConstraintSql();
```

Príklad: Použitie skriptu pri generovaní SQL

Funkcia z príkladu sa vloží do sekcie *script* v danom pravidle. Opäť pripomíname, že formát XML používa niektoré znaky v rámci svojho syntaxu, preto nie je dovolené používať ich v texte. Nahradzovať ich musíme sekvenciou iných znakov, napríklad znak " nahradíme sekvenciou *"*. Celkovo bude pravidlo vyzerať nasledovne:

```
...
<object name="ora_constraint">
  <newRules>
    ...
  </newRules>
  <obsoleteRules>
    <rule name="constraint-drop">
      <condition>
        <stateItem operator="isNot" objectName="ora_table">
          <state>obsolete</state>
        </stateItem>
      </condition>
      <script>
        function dropConstraintSql() {
          var s = &quot;ALTER TABLE &quot; + currentObj.parent.NAME +
               &quot; DROP CONSTRAINT &quot;
               + currentObj.NAME + &quot;\n\n&quot;;
          addSqlText(s, &quot;dropConstraints&quot;);
        }

        dropConstraintSql();
      </script>
    </rule>
    ...
  </obsoleteRules>
  ...
</object>
...
```

Príklad: Pravidlo používajúce skript v kontexte konfiguračného súboru

Je pomerne pracné ručne vytvárať konfiguračný súbor, ktorého časti sme si postupne popísali. Objektov a pravidiel je relatívne veľké množstvo. Preto sme sa rozhodli vytvoriť grafické prostredie, ktoré uľahčí vytvorenie konfigurácie, ale aj definovanie pripojení k databázam, spustenie porovnania databázových schém, zisťovanie informácii o ich štruktúre a spúšťanie SQL príkazov. Grafické prostredie stručne popíšeme v nasledujúcej kapitole.

5.4 Implementácia v Jave

V tejto sekcii stručne naznačíme, akým spôsobom je aplikácia implementovaná v Jave. Popíšeme len základné časti a hlavné triedy, pre lepšiu orientáciu odporúčame preštudovať dokumentáciu JavaDoc k aplikácii *compare*, ktorú čitateľ nájde na priloženom CD.

Vstupnou triedou do konzolovej aplikácie *compare* je trieda *Application*, ktorá obsahuje aj metódu *main*. Úlohou tejto triedy je získanie argumentov, načítanie XML súborov a ich uloženie do pamäte. Všetky tri vstupné XML súbory sú v pamäti reprezentované objektmi z balíkov `com.tnix.compare.jaxb.*`. Takto konvertovaný vstup sa odovzdá triede *CompareApplication*, ktorej metóda execute vhodnými volaniami metód ostatných tried vykoná celý proces porovnania a generovania výstupu. Grafický klient *DbStudio*, ktorý popisujeme v ďalšej kapitole, využíva na inicializáciu procesu porovnávania schém priamo triedu *CompareApplication*.

Trieda *Compare* obsahuje implementáciu algoritmov 1 a 2 z tretej kapitoly. Jej výstupom je diff strom, ktorý sa následne spracováva triedami *GenerateSql* a *GenerateXmlAndText*. Prvá trieda prechádza diff strom a podľa definovaných pravidiel vytvára výsledný SQL skript, druhá na základe toho istého princípu vytvorí XML a textový výstup.

Za zmienku stoja ešte triedy, ktoré sa nachádzajú v balíku `com.tnix.compare.javascript`. Sú to triedy, ktoré podporujú integráciu skriptovacieho jazyka *JavaScript* do aplikácie. Trieda *BaseJs* implementuje základné funkcie, ktoré môžeme v skriptoch používať, trieda *ObjectTypeWrapper* zapúzdruje akýkoľvek databázový objekt. Užívateľ môže vhodným volaním funkcií implementovaných práve v tomto objekte voľne prechádzať celý diff strom, alebo sa dotazovať na akúkoľvek vlastnosť databázového objektu.

6 GRAFICKÉ ROZHRANIE

Program *DbStudio* tvorí grafické rozhranie k synchronizácii databázových schém. Je to aplikácia napísaná v jazyku Java s využitím knižnice *JFC/Swing*. Jej základná funkcia je poskytnúť užívateľovi kontrolu nad použitím JDBC vrstvy. V prehľadnom grafickom prostredí môže nahrávať vlastné JDBC drivery z JAR súborov a používať ich na pripojenie k definovaným databázam. *DbStudio* umožňuje prezerať štruktúru schém databáz, vykonávať dotazy a zobrazovať ich výsledky. Zrejme najzaujímavejšou vlastnosťou aplikácie je však integrácia so synchronizačným programom databázových schém. Užívateľ môže vybrať schémy, ktoré sa majú porovnávať. *DbStudio* dokáže automaticky nakonfigurovať *Schemagic* a získať údaje o obidvoch databázových schémach. V editore užívateľ vytvorí konfiguráciu synchronizácie, *DbStudio* následne schémy porovná a výsledok prehľadne zobrazí v strome. Zároveň používateľovi ponúkne ostatné druhy výstupu – vo forme textu, XML a SQL skriptu.

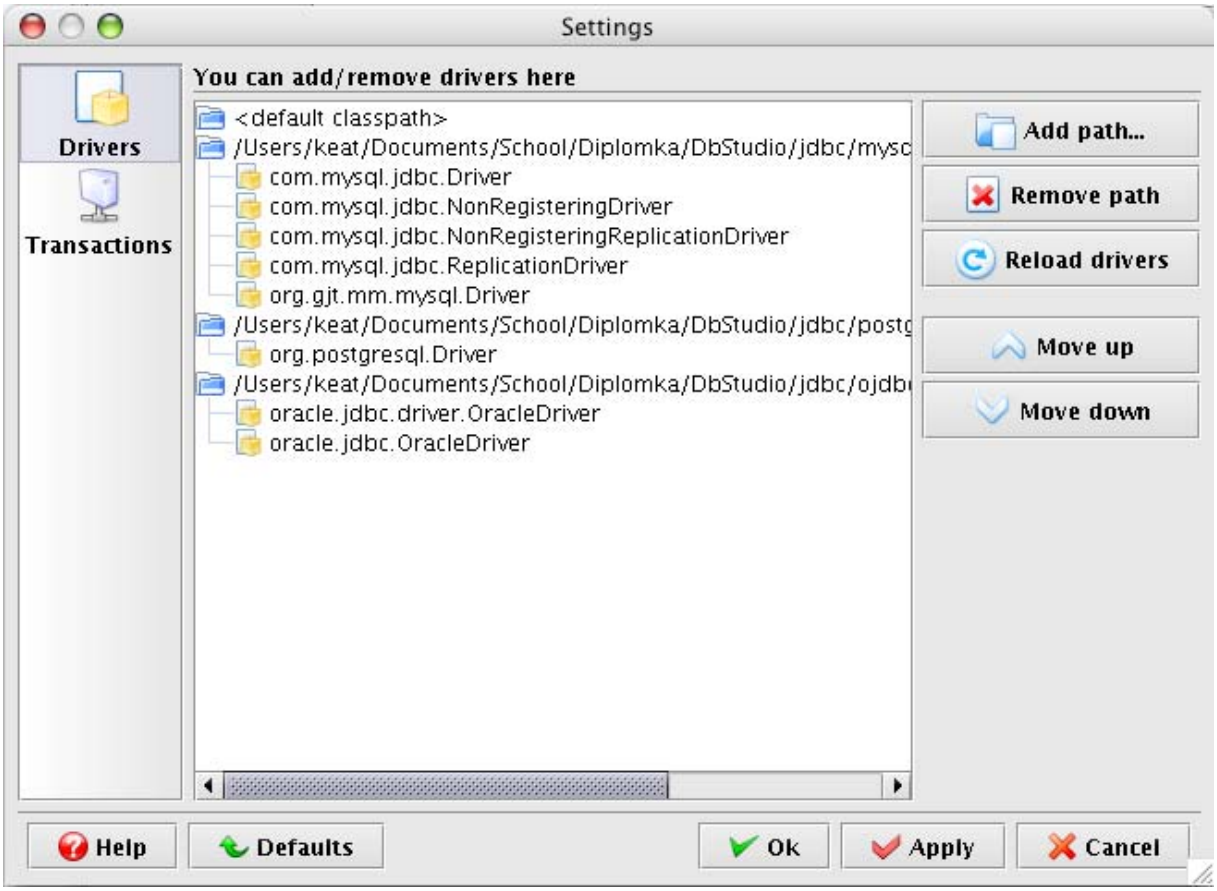
DbStudio je komplexná aplikácia. O jej veľkosti svedčí aj to, že ju tvorí približne 250 negenerovaných Java tried, ktoré spolu obsahujú zhruba 35 000 riadkov kódu. Činnosť tohto grafického prostredia si priblížime v nasledujúcich kapitolách, podrobný návod na ovládanie DbStudia sa však kvôli svojmu rozsahu do tejto práce nezmestil a je priložený na CD.

6.1 Dynamické nahrávanie JDBC driverov

JDBC drivery sú spravidla zahrnuté do balíčka skompilovaných tried jazyka Java. Úlohou drivera je pripojiť sa ku konkrétnemu typu databázy a to buď natívne, alebo s využitím iného rozhrania (napr. ODBC). Po vytvorení pripojenia môže program cez JDBC vrstvu posielať dotazy na dáta a sprostredkovať výsledky späť používateľovi. Pretože vrstva JDBC je dnes často používaná, ku každému významnému databázovému stroju existuje jeden, alebo viac druhov JDBC driverov, ktoré sa môžu líšiť vo funkcionalite.

JDBC drivery sú väčšinou dodávané v archíve JAR, zriedkavo sú triedy uložené priamo v adresári. Ak sú JAR archívy uvedené v ceste (tzv. CLASSPATH) pri spustení aplikácie, sú drivery nahraté automaticky. V opačnom prípade ich musíme do programu zaregistrovať iným spôsobom.

Na registráciu JDBC driverov slúži v aplikácii dialóg, v ktorom špecifikujeme cesty k JAR archívom a určíme poradie, v akom sa v nich bude vyhľadávať. Program JDBC drivery vyhľadá a zobrazí ich, následne ich môžeme používať. Konfigurácia *DbStudia* sa ukladá na disk vo forme XML súbora, takže registráciu JDBC driverov stačí previesť raz, pri nasledujúcom spustení sa do aplikácie zaregistrujú drivery z určených ciest automaticky.

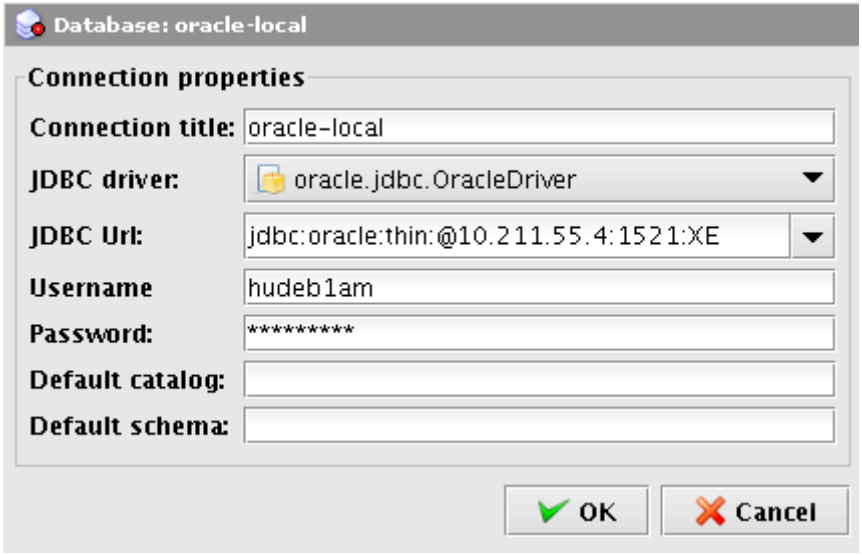


Obr.: Registrácia JDBC driverov v aplikácii *DbStudio*

6.2 Pripojenie k databáze

Ak je v *DbStudiu* zaregistrovaný príslušný driver, môžeme sa pripojiť k databáze. Budeme na to potrebovať niekoľko údajov. JDBC URL je spôsob, ako sa v JDBC identifikuje databáza. Podľa tvaru JDBC URL sa použije príslušný driver, ktorý potom nadviaže spojenie s databázou [21]. Užívateľ musí uviesť aj prihlasovacie údaje

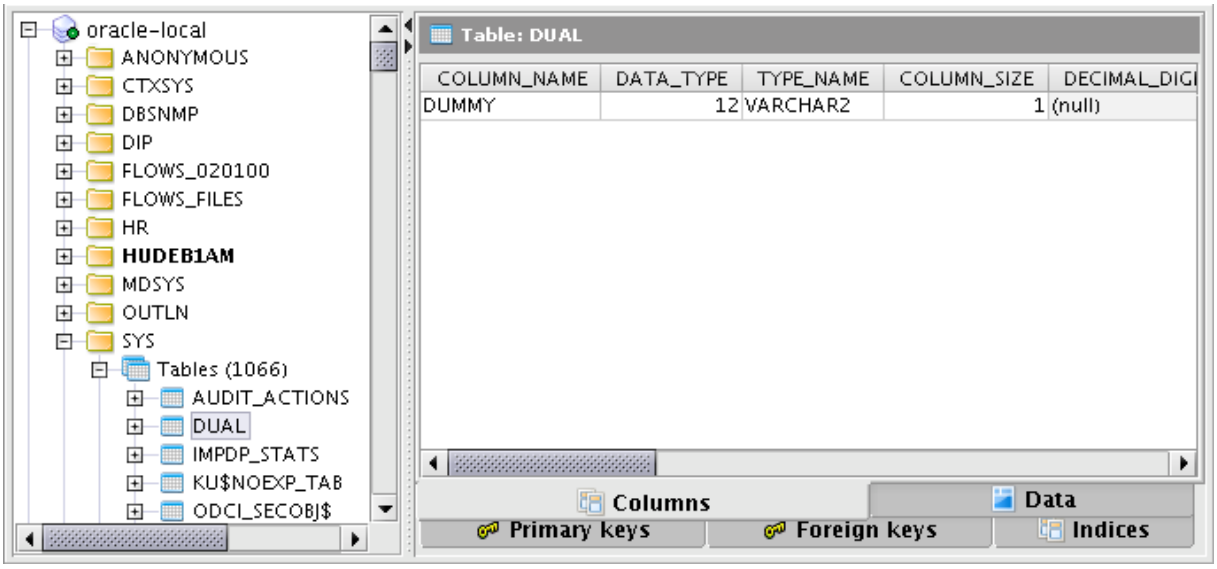
(prihlasovacie meno a heslo). Po vyplnení údajov sa *DbStudio* pripojí k databáze a to v samostatnom vlákne.



Obr.: Definovanie informácii o databáze, ku ktorej sa pripájame

6.3 Zobrazovanie informácií o databázach

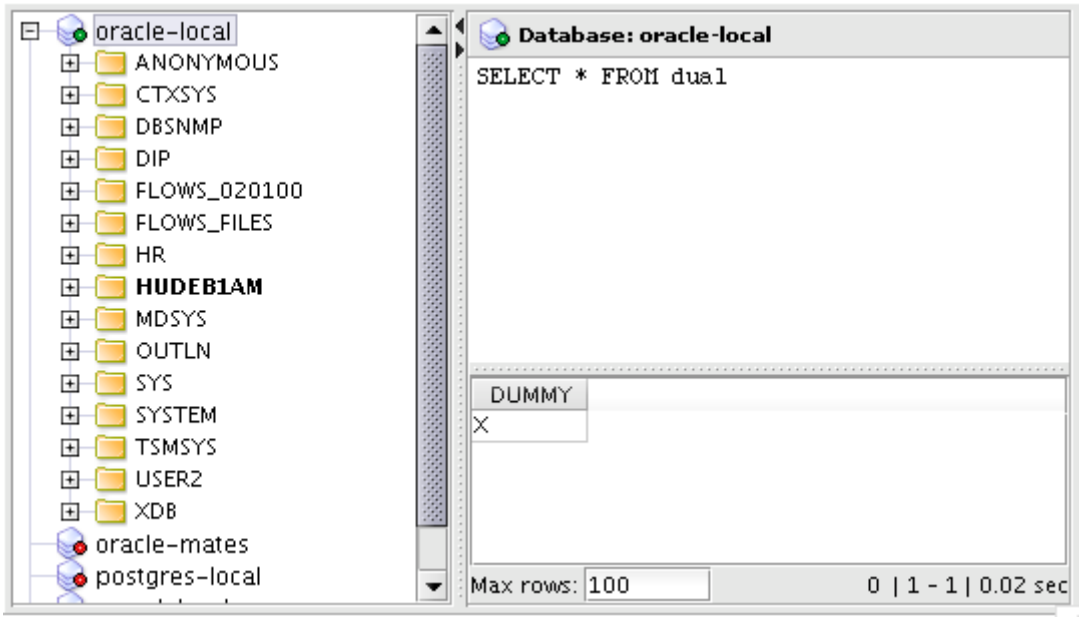
DbStudio poskytuje základné informácie o pripojených databázach a ich schémach. Databázové objekty sú zoradené do stromu a po kliknutí na konkrétny objekt užívateľ vidí informácie o objekte, ako napríklad zoznam stĺpcov tabuľky, stĺpce primárneho kľúča, alebo samotné dáta v tabuľke.



Obr.: Zobrazenie informácii o databázovom objekte

6.4 Vykonávanie dotazov

Prostredníctvom DbStudia môže užívateľ vykonávať ľubovoľné dotazy do databázy. V prípade, že sa jedná o dotazy typu SELECT, výsledok sa zobrazí pod dotazom. K dispozícii je aj ovládanie transakcií, štandardný mód je AUTOCOMMIT, to sa však dá jednoducho zmeniť. Transakcie sa potom štartujú manuálne a ukončujú sa tlačidlami COMMIT, alebo ROLLBACK. K dispozícii je aj história SQL príkazov.



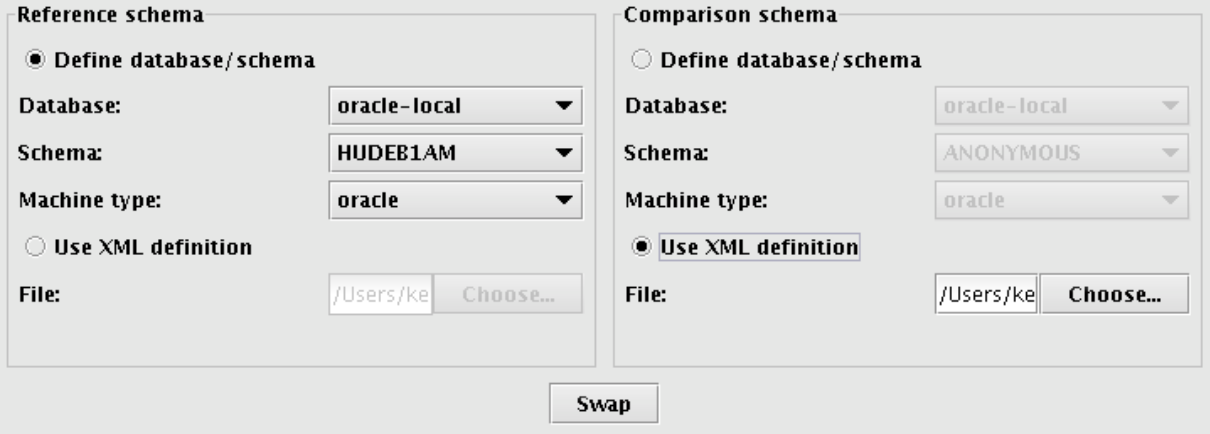
Obr.: Vykonanie dotazu

6.5 Synchronizácia databázových schém

Najdôležitejšou časťou grafického rozhrania je samozrejme ovládanie samotnej synchronizácie. Namiesto ručného spúšťania aplikácie *Schemagic* a generovania vstupu pre konzolovú aplikáciu na porovnávanie schém môžeme celú synchronizáciu (počnúc výberom databáz až po výstupné súbory) zrealizovať v *DbStudiu*. Celý proces prebieha v krokoch – užívateľ najprv vyberie konkrétne schémy, následne sa mu stromy týchto schém graficky znázornia. V ďalšom kroku vytvorí (prípadne načíta) konfiguračný súbor, podľa nastavení porovná schémy a vygeneruje výstupné súbory, ktoré potom zobrazí.

6.5.1 Načítanie dát

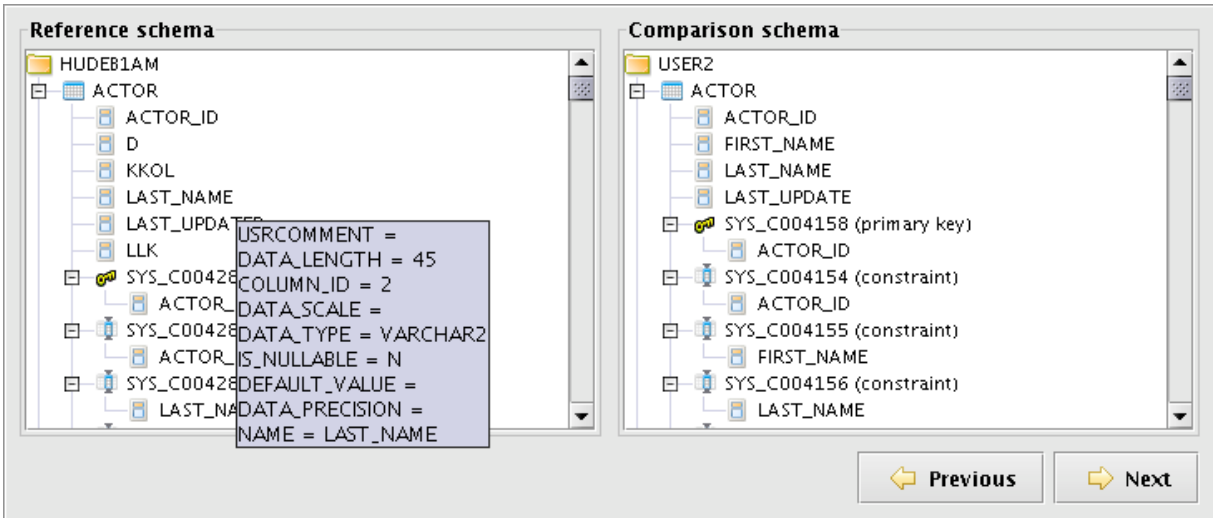
Dáta o databázových schémach sa do programu môžu dostať dvoma spôsobmi. Buď z ponuky vyberieme databázu, ku ktorej je *DbStudio* pripojené a určíme konkrétnu schému, alebo zvolíme XML súbor (výstup programu *Schemagic*), v ktorom sú už údaje načítané.



Obr.: Špecifikácia databázových schém, ktoré porovnávame

6.5.2 Zobrazenie stromov schém

Informácie o databázových schémach načítané buď z XML súboru, alebo priamo z databázy sa užívateľovi zobrazia v dvoch oddelených častiach vo forme stromu. Každý databázový objekt má nastavené hodnoty vlastností, užívateľ ich vidí pri prechode kurzora nad objektom.

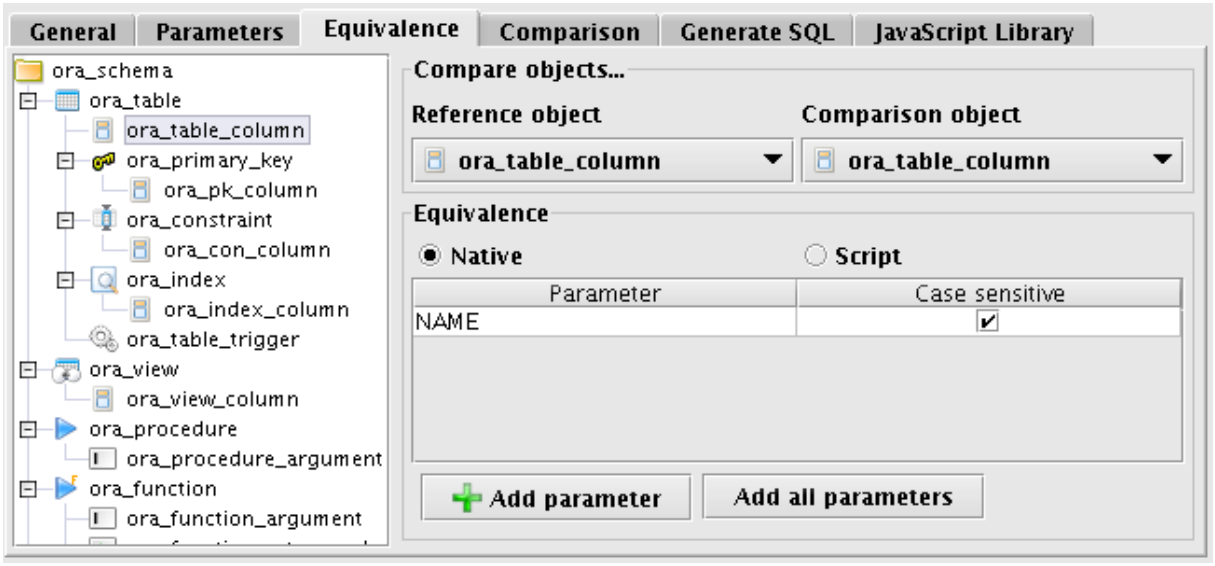


Obr.: Stromy referenčnej a porovnávanej schémy

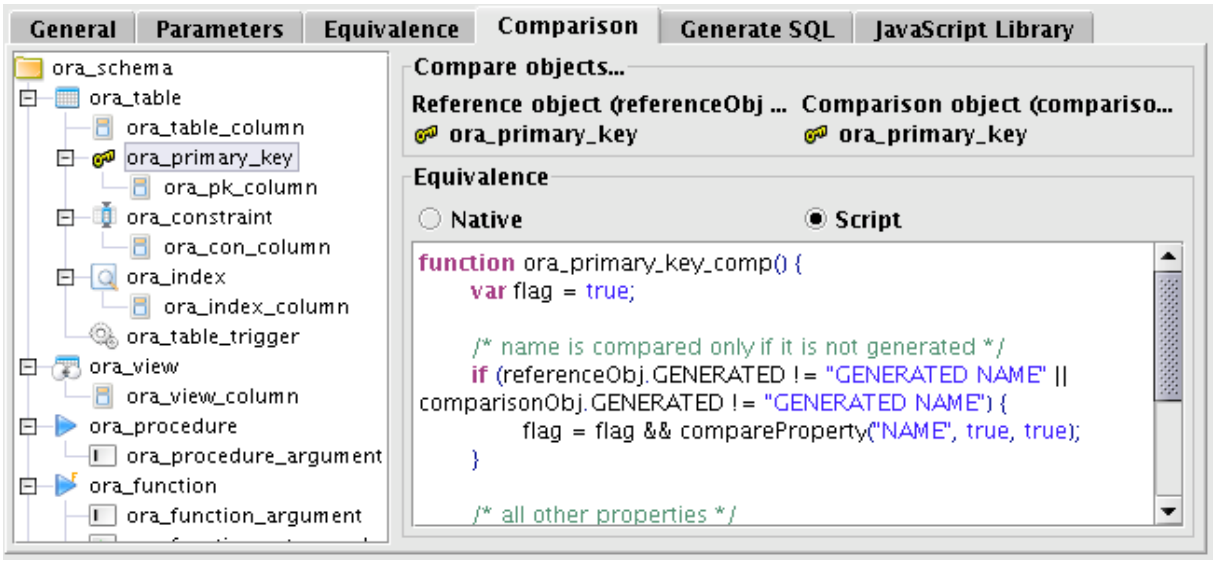
6.5.3 Konfigurácia porovnania

Konfiguráciu porovnania sme podrobne popísali v predchádzajúcich kapitolách. Vytvorenie konfiguračného súboru manuálne by bolo zdĺhavé a pracné, preto je odporučenou metódou na jeho tvorbu použitie grafického rozhrania. Po nastavení všetkých parametrov sa dajú nastavenia uložiť do XML formátu a neskôr opätovne z tohto súboru načítať. Vytvorený XML súbor sa samozrejme dá použiť aj v konzolovej verzii aplikácie.

Konfigurácia ekvivalencie databázových objektov je oddelená od konfigurácie porovnania vlastností dvoch objektov (dva rôzne *taby*). V obidvoch prípadoch máme na výber, či chceme použiť natívny spôsob, alebo skript jazyka JavaScript.



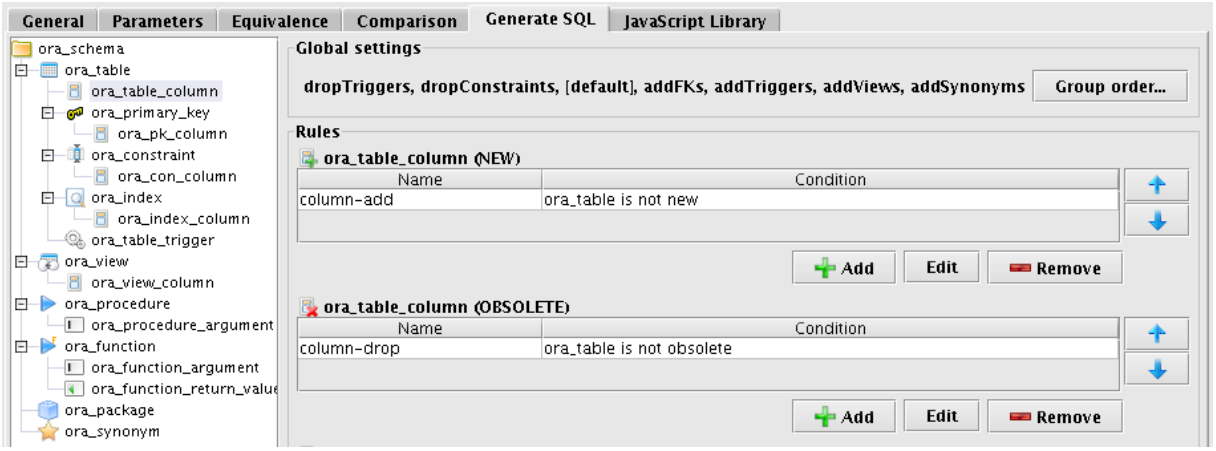
Obr.: Konfigurácia ekvivalencie – používame natívny spôsob



Obr.: Konfigurácie porovnania vlastností – skript jazyka JavaScript

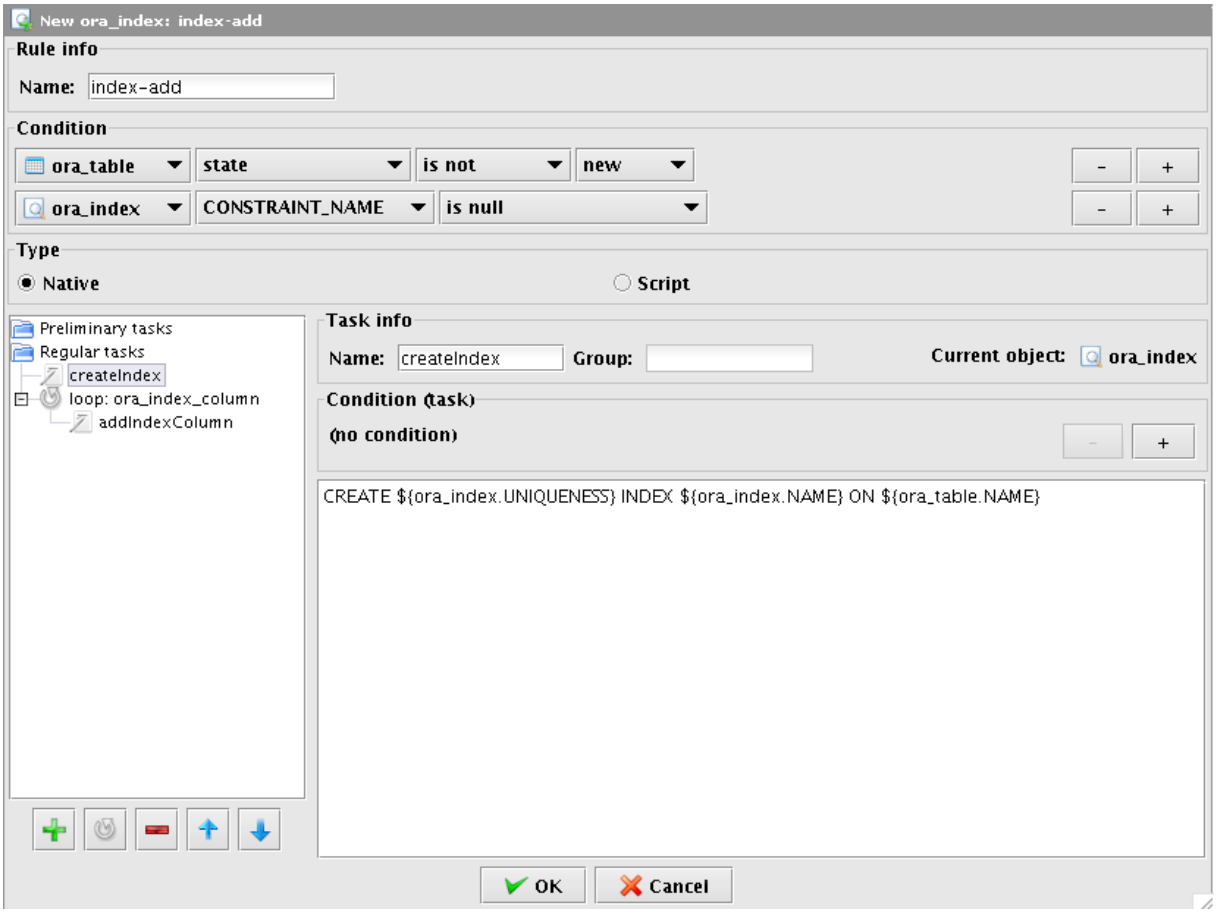
6.5.4 Konfigurácia generovania SQL skriptu

Pre každý databázový objekt, zobrazený v strome vľavo môžeme definovať pravidlá a zaradiť ich do troch skupín – pravidlá, ktoré reagujú na objekty so stavom *new*, *obsolete* a *changed*. V aplikácii môžeme určovať poradie pravidiel a definovať skupiny SQL príkazov.



Obr.: Pravidlá na generovanie SQL pre typ *ora_table_column*

Po pridaní nového pravidla potrebujeme editovať jeho základné údaje a pridať úlohy, či už štandardné, alebo také, ktoré sa vykonajú v rámci definovaných skupín. Súčasťou definície pravidla môže byť podmienka, ktorá je zložená z niekoľkých častí spojených logickým AND. Podmienka sa dá editovať priamo v okne, výberom z príslušnej ponuky. Do pravidla pridávame úlohy – jednoduché a cykly. Každá jednoduchá úloha môže obsahovať ďalšiu podmienku a výstupný text. V texte sa môžu nachádzať premenné a volania skriptovacích funkcií.



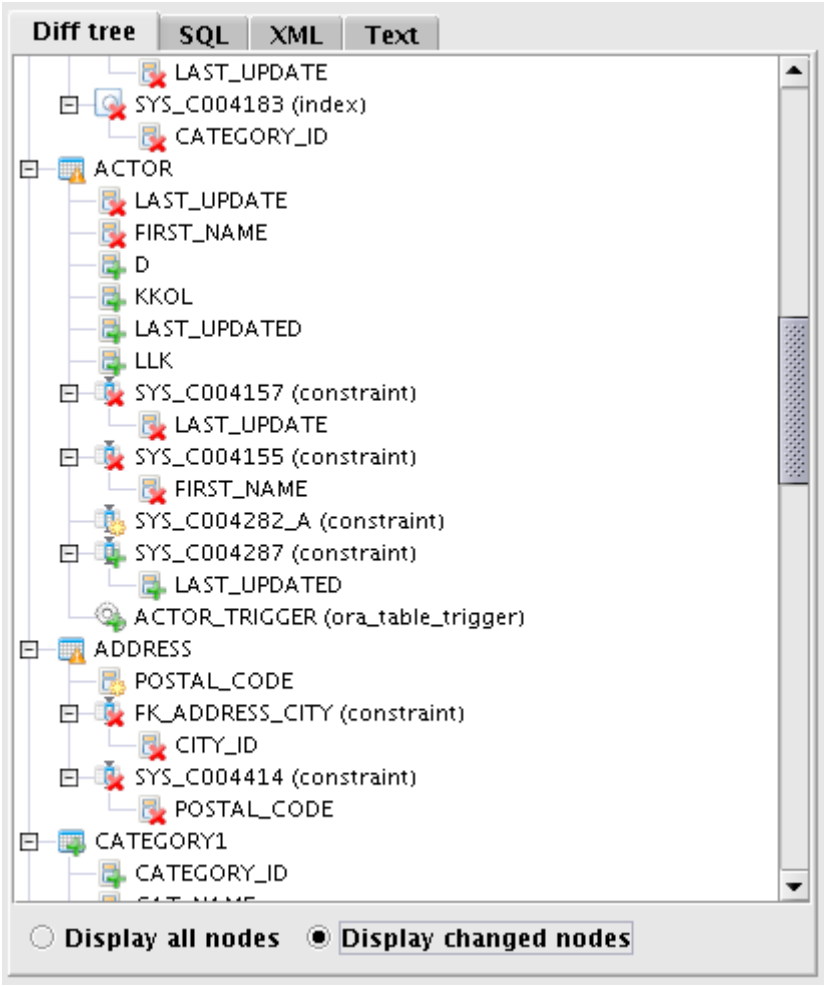
Obr.: Editácia pravidla *index-add*

6.5.5 Zobrazenie výstupu

V poslednom kroku *DbStudio* užívateľovi zobrazí štyri typy výstupu. Prvý typ môže získať len prostredníctvom grafického rozhrania, ostatné tri sú zhodné s výstupom z konzolovej aplikácie. Všetky typy výstupu sa vytvoria v jednom kroku a užívateľ medzi nimi prepína pomocou záložiek.

6.5.5.1 Strom zmien (diff strom)

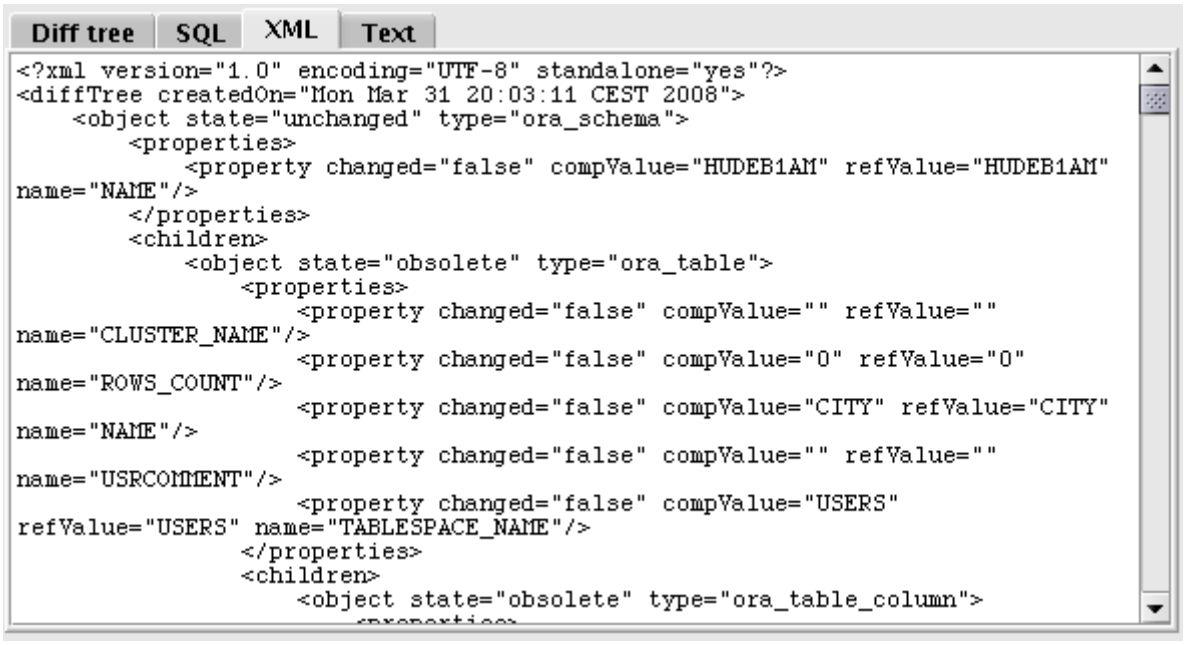
Strom zmien znázorňuje databázové objekty a ich stav. Zelenou značkou sú vyznačené databázové objekty, ktoré sa vyskytujú v referenčnej, ale nevyskytujú v porovnávanej schéme. Červenou značkou znázorňujeme objekty, ktoré naopak nie sú v referenčnej schéme, ale sú v porovnávanej schéme. Žltá značka znamená, že objekt je aj v jednej, aj v druhej schéme, ale hodnota niektorej vlastnosti je rôzna. Ak je kurzor umiestnený nad grafickou reprezentáciou tohto objektu, zobrazí sa zoznam zmenených vlastností, spolu s hodnotami.



Obr.: Grafická reprezentácia diff stromu

6.5.5.2 XML výstup

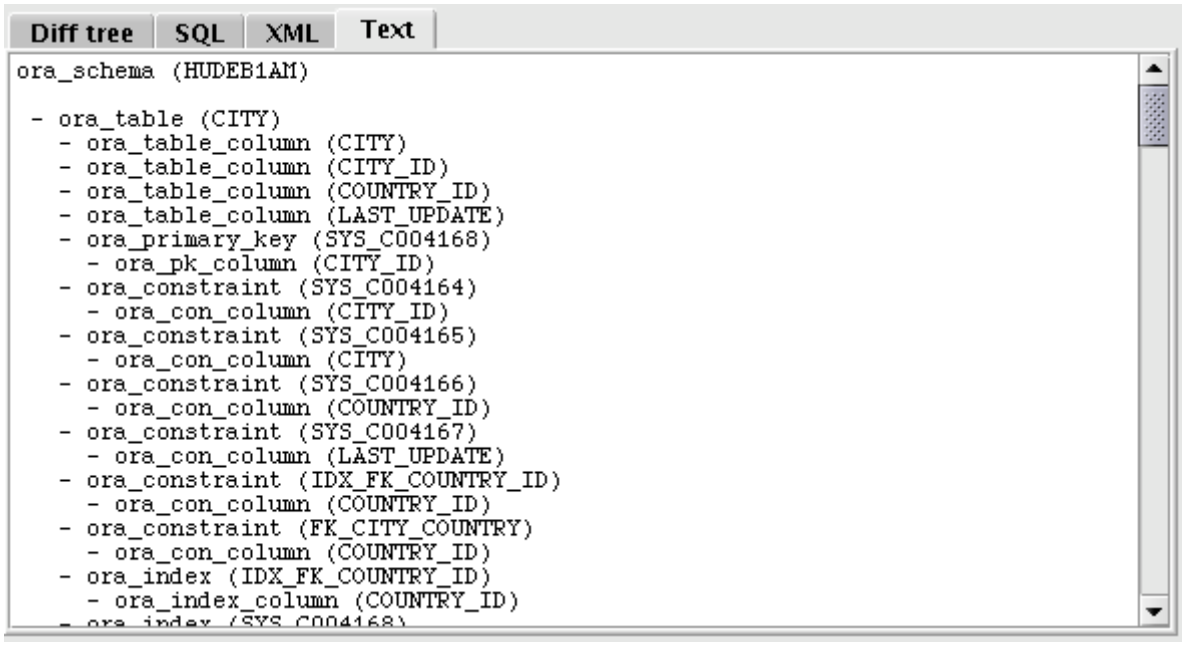
XML transformácia diff stromu je vhodný výstup v prípade, ak chceme údaje ďalej spracovávať v externom programe. XML výstup sa nachádza v príslušnej záložke, je možné ho uložiť do súboru.



Obr.: XML výstup

6.5.5.3 Textový výstup

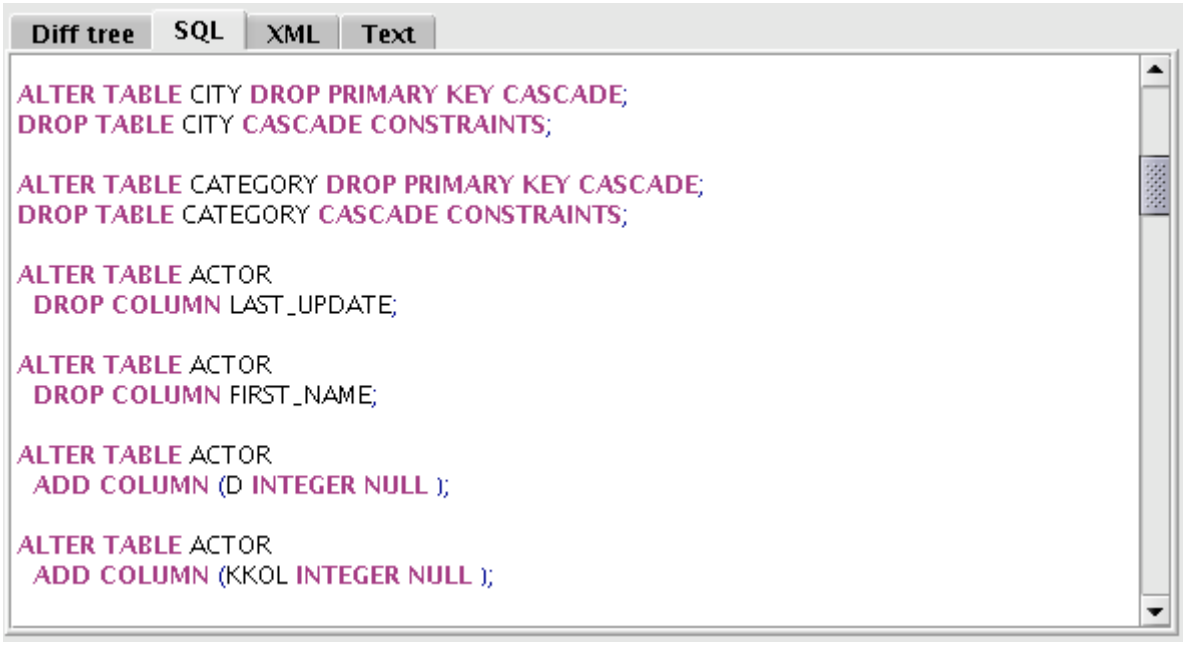
Textový výstup slúži hlavne na archiváciu výstupov programu. V podstate sa jedná o prepis grafickej reprezentácie diff stromu do textu. Užívateľ opäť môže uložiť obsah do súboru.



Obr.: Textový výstup

6.5.5.4 SQL skript

Aplikáciou pravidiel z konfiguračného súboru sa postupným prechádzaním diff stromu vytvorí výsledný SQL skript, užívateľ ho opäť nájde v samostatnej záložke. Text je kvôli prehľadnosti zvýraznený (zvýraznené sú čísla, reťazce a kľúčové slová podľa štandardu ANSI SQL).



Obr.: SQL skript, synchronizujúci porovnávanú schému s referenčnou

7 ZÁVER

Cieľom práce bolo navrhnúť a vytvoriť nástroj umožňujúci porovnanie dvoch databázových schém, ktoré môžu byť súčasťou rôznych databázových strojov. Nástroj mal byť spustiteľný na čo najväčšom počte platforiem a mal byť jednoducho rozšíriteľný na rôzne typy databázových strojov. Ďalšie dôležité vlastnosti zahŕňali hlavne jednoduchosť používania, cielenú predovšetkým na databázových administrátorov, možnosť spúšťať porovnávanie automatizovane pomocou konzolovej aplikácie a niekoľko druhov výstupov pre ďalšie strojové spracovanie a archiváciu.

V analytickej časti práce sme sa venovali algoritmom, ktoré sme neskôr použili pri implementácii. Rozlíšili sme dve fázy porovnávania databázových objektov a podrobne sme sa venovali hlavnej dátovej štruktúre, ktorú používame v celej aplikácii – rozdielovému (diff) stromu. Pomerne veľký priestor sme venovali analýze generovania rôznych druhov výstupu, osobitnú pozornosť si vyžadoval hlavne SQL výstup, pri ktorom sme museli zohľadniť jeho špecifické črty, napríklad poradie SQL príkazov, alebo obsluhu niekoľkých objektov jedným komplexným SQL príkazom.

Pri návrhu sme okrem stanovených požiadaviek dbali najmä na možnosť konfigurácie nástroja. Rôzni používatelia môžu mať odlišné predstavy o tom, ako má porovnávanie prebiehať, preto by sme im mali umožniť nastaviť rôzne parametre porovnávania. V zvláštnych prípadoch užívatelia dokonca majú prístup priamo k interným dátovým štruktúram a to prostredníctvom skriptovacieho jazyka. Práve konfigurovateľnosť v konečnom dôsledku pozitívne ovplyvnila reálnu použiteľnosť aplikácie.

Po analýze sme sa venovali spôsobu implementácie v jazyku *Java*. Popísali sme možnosti použitia natívnych pravidiel, alebo jazyka *JavaScript* vo všetkých fázach porovnávania, objasnili sme, akým spôsobom sa vytvára konfiguračný súbor. V poslednej kapitole sme predstavili aj grafické rozhranie, ktoré tvorí podstatnú časť práce, pretože umožňuje jednoduchú konfiguráciu, spustenie synchronizácie a prehľadné zobrazenie výsledku. Grafický klient umožňuje aj ďalšiu prácu s databázami, takže užívateľ nemusí

siahať po iných nástrojoch a aplikáciu *DbStudio* môže využiť aj na bežnú prácu s databázou ľubovoľného typu.

Aplikáciu sme otestovali na niekoľkých databázových schémach, patriacich databázam typu Oracle a MySQL. Nájdené zmeny vždy zodpovedali skutočnosti, tj. aplikácia našla všetky vykonané zmeny. Výstupný SQL skript závisí hlavne na kvalite konfigurácie. Vzhľadom na to, že aplikácia ponúka širokú škálu možností, ako pomocou definovania pravidiel dosiahnuť želaný výstup, boli sme schopní dosiahnuť v podstate totožný výsledok, ako komerčné nástroje, napr. už spomínaný nástroj *Toad for Oracle* od firmy *Quest*. Užívateľ však na rozdiel od týchto nástrojov nie je odkázaný na vstavanú štandardnú funkcionalitu, ale môže výstup doplniť o rôzne doplnkové SQL príkazy, vlastné komentáre, prípadne ladiace volania. Navyše môže využívať odlišné konfigurácie pre rôzne spôsoby použitia aplikácie. Aplikácia je určená pre pokročilého užívateľa, ktorý synchronizácie vykonáva pomerne často a teda je pre neho výhodné vytvoriť konfiguráciu, ktorá presne zodpovedá jeho požiadavkám.

Domnievame sa, že vďaka variabilite a konfigurovateľnosti je implementovaná aplikácia použiteľná v praxi a je možné ju nasadiť v reálnom prostredí. Okrem toho, že nástroj prehľadne informuje užívateľa o rozdieloch medzi databázovými schémami, dokáže tiež generovať kód, ktorý štruktúry týchto schém zosynchronizuje. Databázový administrátor jeho použitím ušetrí čas a predíde prípadným chybám, ktoré by mohli vzniknúť existenciou dvoch verzií tej istej databázovej schémy, čo bol od začiatku náš hlavný cieľ.

A OBSAH CD

Priložené CD obsahuje implementáciu konzolovej, aj grafickej aplikácie, návod na používanie grafickej aplikácie, samotný text práce a rôzne ďalšie dokumenty. Podrobný popis obsahu CD nosiča sa nachádza v súbore *readme.txt*.

Spustenie konzolovej aplikácie

Na spustenie konzolovej aplikácie budeme potrebovať tri XML súbory, dva z nich získame ako výstup z aplikácie *Schemagic*. Podrobné informácie o tom, ako ich získame, nájde čitateľ v [7]. Vstupné súbory sú:

- XML súbor, popisujúci referenčnú schému (výstup z aplikácie *Schemagic*)
- XML súbor, popisujúci porovnávanú schému (výstup z aplikácie *Schemagic*)
- konfiguračný súbor s nastaveniami

Aplikáciu spustíme príkazom:

```
compare.bat [argumenty] <referencna schema> <porovnavana schema>
<konfiguracny subor>
```

Výstupom aplikácie sú tri súbory, ktoré vzniknú v definovanom adresári a majú prípony *.xml*, *.txt* a *.sql*. Názov súboru je odvodený od dátumu a času, kedy bola konverzia spustená. Môžeme použiť argument `--name <nazov porovnavania>`, ktorý určí názov výstupných súborov (napr. *<názov porovnávania>.sql*, atď.). Adresár môžeme určiť prepínačom `--dir <nazov adresara>`, všetky súbory sa potom uložia do tohto adresára. V prípade, že adresár užívateľ nešpecifikuje, súbory sa uložia do aktuálneho pracovného adresára.

Spustenie grafickej aplikácie DbStudio

Konzolová aplikácia je vhodná pre skúsených používateľov, prípadne na realizáciu automatizovaných porovnaní databázových schém. V ostatných prípadoch odporúčame použiť grafickú aplikáciu *DbStudio*, pomocou ktorej sa stáva celý proces porovnávania

jednoduchý a priamočiary. Podrobný návod na spustenie a používanie aplikácie *DbStudio* sa nachádza na priloženom CD v PDF formáte.

B PODMIENKOVÉ OPERÁTORY

O podmienkach, ktoré používame v pravidlách generovania SQL príkazov sme písali v kapitole 5.3.2.3. V príkladoch sme však uviedli len niekoľko operátorov, ktoré môže užívateľ použiť. V tejto kapitole uvádzame ich plný zoznam.

Testovanie stavu databázového objektu

Operátor	Vysvetlenie
is	Vráti *true*, ak je stav objektu zhodný s uvedeným stavom
isNot	Vráti *true*, ak stav objektu nie je zhodný s uvedeným stavom
isOneOf	Vráti *true*, ak je stav objektu zhodný s jedným z uvedených stavov
isNotAnyOf	Vráti *true*, ak sa stav objektu nerovná žiadnemu z uvedených stavov

Testovanie hodnoty vlastnosti databázového objektu

Operátor	Vysvetlenie
is	Vráti *true*, ak sa vlastnosť rovná uvedenej hodnote
isNot	Vráti *true*, ak sa vlastnosť nerovná uvedenej hodnote
strContains	Vráti *true*, ak hodnota vlastnosti obsahuje podreťazec zhodný s uvedenou hodnotou
isNull	Vráti *true*, ak je vlastnosť objektu prázdna
isNotNull	Vráti *true*, ak je vlastnosť objektu nie je prázdna
isEqualValue	Porovná hodnoty vlastností dvoch objektov, vráti *true*, ak sa zhodujú
isNotEqualValue	Porovná hodnoty vlastností dvoch objektov, vráti *true*, ak sa nezhodujú

Testovanie zmenených vlastností

Operátor	Vysvetlenie
listContains	Vráti true, ak zoznam zmenených vlastností obsahuje špecifikovanú vlastnosť
listDoesNotContain	Vráti *true*, ak zoznam zmenených vlastností neobsahuje špecifikovanú vlastnosť
listContainsAtLeastOne	Vráti *true*, ak zoznam zmenených vlastností obsahuje aspoň jednu vlastnosť zo špecifikovaného zoznamu
listDoesNotContainAllOf	Vráti *true*, ak zoznam zmenených vlastností neobsahuje ani jednu vlastnosť zo špecifikovaného zoznamu

POUŽITÁ LITERATÚRA

[1] J. Pokorný. *Dotazovací jazyky*. Prague: Karolinum press, 2002.

[2] Wikipedia. *Database management system* [online]. [cit. 15.4.2008]. http://en.wikipedia.org/wiki/Database_management_system.

[3] Wikipedia. *SQL* [online]. [cit. 15.4.2008]. http://en.wikipedia.org/wiki/SQL, 2008.

[4] Digital Equipment Corporation. *Information Technology - Database Language SQL (Proposed revised text of DIS 9075)*. http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt.

[5] International Organization for Standardization. *ISO/IEC 9075-1:1999 (SQL 1999 standard)*.

[6] A. Eisenberg. *SQL:2003 has been published*. SIGMOD Record, Vol. 33, No. 1, March 2004.

[7] M. Nagy. *Synchronisation of relational schemas*. Master's thesis, Faculty of Mathematics and Physics, Charles University, Prague, 2005.

[8] Quest Software. *Toad® for Oracle* [online]. [cit. 15.4.2008]. http://www.quest.com/toad-for-oracle/.

[9] J. Clark. *XSL Transformations (XSLT), Version 1.0, 1999* [online]. [cit. 15.4.2008]. http://www.w3.org/TR/xslt.

[10] EMS Database Management Solutions. *EMS DB Comparer for MySQL* [online]. [cit. 15.4.2008]. http://www.sqlmanager.net/en/products/mysql/dbcomparer.

[11] DB Solo LLC. *DB Solo - schema comparison tool* [online]. [cit. 15.4.2008]. http://www.dbsolo.com/schema_comparison.html.

[12] Sun Developer Network. *Java SE technologies - Database* [online]. [cit. 15.4.2008]. http://java.sun.com/javase/technologies/database/.

[13] ECMA. *ECMAScript Language Specification, Standard ECMA-262*. 3rd Edition, December 1999.

[14] Mozilla.org. *Rhino: Javascript for java* [online]. [cit. 15.4.2008]. http://www.mozilla.org/rhino/.

[15] Wikipedia. *XML* [online]. [cit. 15.4.2008]. http://en.wikipedia.org/wiki/XML.

[16] JDOM.org. *JDOM* [online]. [cit. 15.4.2008]. http://www.jdom.org/.

[17] Java.net. *JAXB Reference Implementation Project* [online]. [cit. 15.4.2008]. https://jaxb.dev.java.net/.

[18] D. Fallside, P. Walmsley. *XML Schema Part 0: Primer Second Edition* [online]. [cit. 15.4.2008]. http://www.w3.org/TR/xmlschema-0/.

[19] Sun Microsystems, Inc. *JavaTM 2 Platform Standard Ed. 5.0 – String* [online]. [cit. 15.4.2008]. http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html.

[20] N. Boyd. *Scripting java* [online]. [cit. 15.4.2008]. http://www.mozilla.org/rhino/ScriptingJava.html.

[21] Sun Microsystems, Inc. *The Java Tutorials: Establishing a Connection* [online]. [cit. 15.4.2008]. http://java.sun.com/docs/books/tutorial/jdbc/basics/connecting.html.