VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

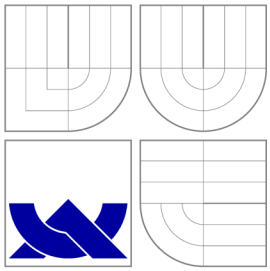
AUTOMATICKÁ TVORBA OBSAHU DATABÁZE SQL PRO PODPORU TESTOVÁNÍ

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

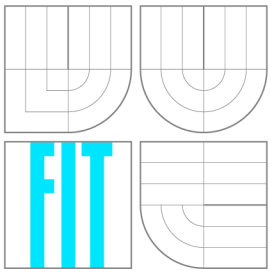
AUTOR PRÁCE
AUTHOR

ALICE MINÁŘOVÁ

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

AUTOMATICKÁ TVORBA OBSAHU DATABÁZE SQL PRO PODPORU TESTOVÁNÍ

A TOOL FOR AUTOMATIC GENERATION OF SQL DATABASE CONTENT FOR SOFTWARE TESTING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — ALICE MINÁŘOVÁ
AUTHOR

VEDOUCÍ PRÁCE — Ing. ALEŠ SMRČKA, Ph.D.
SUPERVISOR

BRNO 2014

Abstrakt

Tato práce se zabývá navržením a implementací dvojice nástrojů pro generování dat za účelem testování. První nástroj analyzuje textový výstup databáze PostgreSQL a vytváří konfigurační soubor v nově navrženém jazyce, který popisuje, jakým způsobem se má vygenerovat obsah databáze. Druhý nástroj na základě tohoto souboru generuje skript SQL dotazů naplňujících cílovou databázi. Uživatel může přizpůsobit generovaná data vlastním požadavkům modifikací konstrukcí v doménově specifickém jazyce. Tento jazyk byl navržen tak, aby byl zásah do konfiguračního souboru pro uživatele rychlý a intuitivní. Část práce se zabývá popisem práce s tímto jazykem. Nástroje byly testovány na uměle navržených databázích i na databázi reálného systému Drupal. Jejich řízení je zcela obslouženo přes příkazový řádek, jsou tedy vhodné k použití při automatizaci.

Abstract

This thesis follows up a design and implementation of a set of two tools for testing data generating. The first tool analyzes PostgreSQL databse text output and creates a configuration file in a newly designed language that describes how the database content should be generated. Based on this file the second tool generates a SQL script to fill the target database. User can adjust the generated data to their own requirements by modifying the configuration file written in a domain-specific language. The language was designed to make possible adjustments quick and intuitive. The thesis also describes how this language should be handled. The two tools were tested on several artificially created databases and also on a real system database of Drupal. The tools are both operated via the command line which makes them suitable for usage in automation.

Klíčová slova

SQL, generátor dat, plnění databáze, testování, automatizace, PostgreSQL, Python, doménově specifický jazyk

Keywords

SQL, Data Generator, Database Filling, Testing, Automatisation, PostgreSQL, Python, Domain-Specific Language

Citace

Alice Minářová: Automatická tvorba obsahu databáze SQL pro podporu testování, bakalářská práce, Brno, FIT VUT v Brně, 2014

Automatická tvorba obsahu databáze SQL pro podporu testování

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracovala samostatně pod vedením pana Ing. Aleše Smrčky, Ph.D. Odborné konzultace probíhaly ve spolupráci s panem Mgr. Matejem Kollárem ze společnosti Red Hat. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

.....
Alice Minářová
20. května 2014

Poděkování

Na tomto místě bych ráda poděkovala vedoucímu své práce, panu Ing. Aleši Smrčkovi, Ph.D, a panu Mgr. Mateji Kollárovi za odborné konzultace, rady a tipy. Dále bych chtěla poděkovat panu Ing. Zbyňku Křivkovi za konzultaci ohledně gramatiky a parsování navrženého doménově specifického jazyka. V neposlední řadě chci poděkovat také svému muži za pomoc při seznamování se systémem Fedora a rady pro svou první větší práci v jazyce Python.

Obsah

Kapitola 1

Úvod

Informační technologie v současné době udávají tempo naší společnosti. S jejich příchodem na scénu se pojí i vznik databázových systémů umožňujících ukládání a správu dat. Databáze jsou dnes nedílnou součastí téměř každé aplikace a jejich testování tedy pochopitelně hraje důležitou roli při ladění celého systému. Má-li však být funkčnost ověřena dostatečně, musí být databáze naplněna dostatečným množstvím vzorových dat. Ruční realizace je časově náročná, lidský faktor do ní nevyhnutelně zanáší chyby a její použitelnost rapidně klesá s rostoucím objemem testovacích dat. Vzniká potřeba automatického plnění databáze.

Smyslem této bakalářské práce je navrhnout, implementovat a zdokumentovat dvojici nástrojů, které uživateli umožní naplnit databázi zvoleným množstvím testovacích dat. První nástroj sloužící k analýze zdrojové databáze vygeneruje mezisoubor definující strukturu tabulek a typ dat, kterými mají být plněny. Mezisoubor bude psán v doménově specifickém jazyce vytvořeném pro tento účel a uživatel bude mít možnost jeho jednotlivé části upravovat podle vlastního uvážení, případně ho zcela vytvořit na základě uvedené gramatiky. Tento soubor bude vstupem druhému nástroji, který na jeho základě provede vygenerování příslušného plnicího skriptu.

Tato práce se nejprve věnuje existujícím řešením problematiky generování dat, jejich analýzou a zhodnocením (kapitola 2). Získané informace poté uvádí do srovnání se zamýšleným provedením plánovaných nástrojů, jejichž samotný návrh následuje (kapitola 3). Součástí této části je také formální návrh doménově specifického jazyka (podkapitola 3.2). Práce se dále zaměřuje na popis implementace obou nástrojů a algoritmů použitých pro řešení konkrétních problematik (kapitoly 4 a 5). Testováním implementovaných řešení obou nástrojů se zabývá kapitola 7. Shrnutím průběhu a výsledků bakalářské práce je tento text zakončen v kapitole 8.

Kapitola 2

Analýza existujících řešení na poli generování testovacích dat

Problematika tvorby testovacích dat není v oboru novinkou, dříve nebo později na ni narazí většina databázových vývojářů. V současné době existuje několik nástrojů, které nabízí generování požadovaných dat a různou míru jejich přizpůsobení požadavkům uživatele.

Mezi tyto nástroje patří:

- **Red Gate – SQL Data Generator**
- **Datanamic – Data Generator for MySQL**
- **Microsoft – Visual Studio Premium**
- **Aplikace generatedata.com**

2.1 Red Gate – SQL Data Generator

Komerční řešení z dílny společnosti **Red Gate** se nazývá **SQL Data Generator**. Nástroj nabízí množství vestavěných generátorů a možnost specifikace generátorů vlastních na základě regulárních výrazů, pythonovských skriptů apod. Uživatel také může dodat externí zdroj dat, která mají být použita k plnění.

SQL Data Generator je ovládán prostřednictvím grafického uživatelského rozhraní, umožňuje však také ovládání přes příkazový řádek, čímž podporuje možnou automatizaci.

Nástroj je implementován pro použití na platformách Windows a databázových serverech Microsoft SQL [12].

2.2 Datanamic – Data Generator for MySQL

Další komerční řešení pochází od vývojářů společnosti **Datanamic**. Nabízí podobné vlastnosti jako výše popsaná varianta od **Red Gate**: vestavěné generátory, tvorbu vlastních generátorů, zajištění referenční integrity, vlastní textové banky jmen, adres, měst, ulic apod. Kromě toho poskytuje uživateli náhled na vzorek dat, která produkuje zvolený generátor. K dispozici je rovněž analýza zdrojové databáze, na jejímž základě nástroj sám přednastaví nejvhodnější generátory. Tyto ovšem uživatel může podle potřeby měnit.

Nástroj je navržen pro práci s databázovými servery MySQL, Oracle, Microsoft SQL či PostgreSQL na systémech Windows. Rovněž umožňuje ovládání přes příkazový řádek [4].

2.3 Microsoft – Visual Studio Premium

Podporu pro generování dat nabízí také **Visual Studio Premium**, produkt společnosti **Microsoft**. Poskytuje podobné vlastnosti jako předcházející nástroje, co se týče zabudovaných generátorů a možnosti tvorby vlastních.

Primárně pracuje pro servery Microsoft SQL, podpora jiných serverů se odvíjí od podpory v konkrétních zásuvných modulech [8].

2.4 Aplikace generatedata.com

Webová aplikace **generatedata.com** nabízí několik základních generátorů dat (jména, e-maily, věty atd.). Vygenerované hodnoty tato aplikace podle volby uživatele zasadí do konstrukcí výstupního formátu (těmi jsou např. Excel, JSON, HTML, CSV, PHP a také SQL).

Generatedata.com umožňuje výběr z několika databázových systémů, pro které vygeneruje výsledný skript. Těmito systémy jsou MySQL, PostgreSQL, SQLite, Oracle a Microsoft SQL.

Použití aplikace je zdarma, jedná se o open source nástroj [2].

2.5 Shrnutí významu získaných informací pro realizaci bakalářské práce

Produkty od společností **Red Gate** a **Datanamic** poskytují uživateli velký prostor při specifikaci generátorů dat. Nástroje pracují ve spojení s cílovou databází, čímž umožňují její přímé plnění.

Jejich ovládání je primárně založeno na práci s grafickým uživatelským rozhraním (jsou tedy vhodné pro uživatele preferující manipulaci s nástroji přes GUI). Ovládání nástroje přes příkazovou řádku se omezuje na příkazy k použití již existujícího projektu vytvořeného pomocí GUI. Ovládání generování dat pouze přes příkazový řádek tedy není možné[1].

Generování dat pomocí **Microsoft – Visual Studio Premium** je jednou z mnoha vestavěných funkcí programu, nástroj z toho důvodu není uživatelským rozhraním specializován na tuto činnost. Manipulace s GUI tedy není tak intuitivní jako v prvních dvou případech.

Visual Studio Premium nepodporuje ovládání přes příkazový řádek ani neposkytuje takové množství vestavěných generátorů, jako generátory od **Red Gate** a **Datanamic**.

Všechny výše zmíněné nástroje jsou komerční řešení pracující pouze na systémech Windows. Jedná se o closed source programy.

Aplikace **generatedata.com** poskytuje malé množství generátorů a neumožňuje žádným způsobem přizpůsobit generovaná data potřebám uživatele. Je tedy použitelná spíše v menších projektech a pouze pro tabulky uchovávající konkrétní typy dat.

Díky umístění na webu se jedná o aplikaci multiplatformní, ovšem bez jakékoliv podpory ovládaní přes příkazový řádek. Aplikace je zdarma a její zdrojové kódy jsou k dispozici jako open source.

[1] Vztahuje se na produkt společnosti **Red Gate**. Dokumentace ke generátorům od **Datanamic** se nepodařilo najít. Vzhledem k minimu prostoru, který je na oficiálních stránkách práci přes příkazovou řádku věnován, však lze předpokládat podobný postup jako u **Red Gate**

Kapitola 3

Obecný návrh práce

Nástroje, které budou vyvinuty v rámci této bakalářské práce (společně dále označovány pracovním názvem „Data Farmer“), poskytnou uživateli možnost naprosté kontroly generování přes příkazový řádek (narozdíl od řešení analyzovaných v kapitole 2). Řešení bude použitelné na serverech PostgreSQL, implementované v jazyce Python 2, čímž by měla být zajištěna přenositelnost mezi různými platformami. Data Farmer by měl nabízet podobné možnosti přizpůsobení generovaných dat jako placené nástroje, bude však dostupný zdarma jako open source.

Jak již bylo naznačeno v úvodu, Data Farmer se bude skládat ze dvou nástrojů. Vhodnost tohoto rozdělení tkví především v možnosti uživatele zasáhnout do procesu větší měrou, než je mu umožněno prostřednictvím přepínačů. Tento zásah bude možný úpravou textového mezisouboru, výstupu prvního nástroje, psaného v doménově specifickém jazyce (DSJ), který bude navržen v rámci této práce. Uživatel bude také moci dodat vlastní textové banky, z nichž bude druhý nástroj brát hodnoty pro generování dat k naplnění tabulek.

Data Farmer by měl podporovat všechny, nebo alespoň většinu datových typů, které PostgreSQL nabízí. Nebude však provádět rozpoznání a analýzu uživatelských datových typů, jejich naplnění musí uživatel provést ve vlastní režii.

3.1 Návrh nástroje pro analýzu zdrojového dumpu a generování souboru v doménově specifickém jazyce

První ze dvou subnástrojů (pracovním názvem „Harvester“ - žnec) bude mít za úkol usnadnit uživateli tvorbu souboru v doménově specifickém jazyce. Bude schopen v tomto jazyce vygenerovat popis plnění cílové databáze na základě uživatelem dodaného databázového dumpu. Analýzou dumpu nástroj získá informace o relacích v databázi a parametrech jejich atributů, které si interně uloží a následně použije při generování mezisouboru.

Aby byl Harvester schopen výše zmíněné údaje získat, bude muset obsahovat částečný SQL parser specializovaný na konstrukce používané generátorem dumpů PostgeSQL.

Existují dva způsoby, kterými lze činnost nástroje pojmout:

1. Parser se bude specializovat na analýzu výhradně těch konstrukcí zdrojového dumpu, které mu umožní vybudovat si interní reprezentaci schématu plněné databáze. Tato reprezentace, řešená použitím vhodných struktur, bude následně použita pro vygenerování mezisouboru, který bude popisovat plnění této databáze.

   Popsaný postup nezahrnuje analýzu již vložených dat, na základě které by Harvester mohl lépe zvolit generovanou metodu plnění. Generovaná plnicí metoda bude

pro všechny atributy shodná a bude se jednat o nejzákladnějš variantu. Počítá se se zásahem uživatele do mezisouboru.

2. Parser bude kromě schématu databáze analyzovat také data, která se už v databázi nachází. Získané údaje použije k odhadu nejvhodnějšího postupu plnění (z dostupných možností), aby generované hodnoty rámcově odpovídaly vlastnostem již vložených hodnot.

   Vysoká úroveň této analýzy by pak mohla uživateli usnadnit práci při výběru vhodných metod a obecně při editaci mezisouboru.

Pro tuto práci byla s ohledem na její rozsah zvolena první varianta. Druhá varianta se však dá považovat za návrh možného budoucího rozšíření.

Vystupní mezisoubor v doménově specifickém jazyce bude odpovídat struktuře tabulek a atributů zdrojové databáze a bude možné ho i bez zásahu okamžitě předat druhému nástroji.

3.2 Návrh doménově specifického jazyka mezisouboru

Doménově specifické jazyky (DSJ) jsou jazyky používané při práci v konkrétních problémových doménách. Podle Martina Fowlera, autora publikace Domain-Specific Languages [6], tyto jazyky zvyšují produktivitu zjednodušením pochopení kódu a při správném navržení usnadňují komunikaci s experty v dané doméně.

Tyto jazyky lze rozdělit na *interní* a *externí*.

Interní doménově specifický jazyk je vytvořen speciálním použitím již existujícího programovacího jazyka. Jeho tvorby lze dosáhnout např. vhodným navržením a pojmenováním funkcí a struktur. Jejich použití následně připomíná vlastní jazyk, vykonání kódu je však řešeno interpretem/překladačem hostitelského jazyka. Interní DSJ jsou častěji používaná varianta DSJ kvůli snadnému použití (nevyžadují navržení gramatiky a parseru). Jsou však svázány konstrukcemi a omezeními hostitelského jazyka.

Externí doménově specifický jazyk je nový jazyk zvlášť vytvořený za účelem řešení konkrétní problematiky. Narozdíl od interních DSJ je pro tyto jazyky nutné navrhnout vhodnou gramatiku s implementovat vlastní parser. Umožňují však větší volnost při tvorbě zamýšlených konstrukcí, nejsou omezeny hostitelským jazykem, pouze schopností programátora v oblasti gramatik a parsování.

Doménově specifický jazyk pro použití v mezisouboru byl navržen jako *externí*. Záměrem při jeho tvorbě byla přehlednost a jednoduchost z pohledu uživatele. Obecná syntaxe zápisu tabulky a jejích atributů bude v DSJ vypadat následovně:

```
TABLE:table_name(number_of_inserts)
::attribute_name
TYPE data_type [(parameters)]
FILL method([parameters])
[CONSTRAINT constraint [(parameters)]]
::attribute_name2
...
```

Identifikátor `table_name` představuje název tabulky, `number_of_inserts` počet řádků, které mají být do této tabulky vloženy. Následují bloky popisující jednotlivé atributy začínají vždy dvěmi dvojtečkami a názvem atributu. Na novém řádku je pak vždy verzálkami

identifikátor vlastnosti atributu a poté samotná vlastnost, v pořadí datový typ, metoda plnění (obojí případně s parametry) a volitelná integritní omezení (`default`, `PRIMARY KEY` atd.).

3.2.1 Gramatika jazyka mezisouboru

Bezkontextová gramatika navrženého doménově specifického jazyka byla popsána pomocí BNF (Backus-Naur Form) [9]. Gramatika včetně popisu tokenů je k dispozici v příloze B.

```
<dsl> ::= <tableBlock> <moreBlocks>

<moreBlocks> ::= <moreBlocks> <tableBlock> | <empty>
<tableBlock> ::= <tableHeader> <attributeBlock> <moreAttributes>
<tableHeader> ::=  TABLE COLON IDENTIFIER LPAREN NUMBER RPAREN <endline>
                |  TABLE COLON IDENTIFIER LPAREN FILLED RPAREN <endline>

<moreAttributes> ::= <moreAttributes> <attributeBlock> | <empty>
<attributeBlock> ::= <attributeName> <dataType> <fillMethod>
                   | <attributeName> <dataType> <fillMethod> <constraintPart>
<attributeName> ::= DOUBLE_COLON IDENTIFIER <endline>


<dataType> ::= TYPE <dtypes> <moreDimensions> <endline>
<dtypes> ::= TYPE_NOPARAM
           | TYPE_1PARAM LPAREN NUMBER RPAREN
           | TYPE_2PARAM LPAREN NUMBER COMMA NUMBER RPAREN
           | TYPE_2PARAM_TIME LPAREN NUMBER COMMA TIMEZONE_PARAM RPAREN

<moreDimensions> ::= <moreDimensions> <oneDimension> | <empty>
<oneDimension> ::= LBRACKET NUMBER RBRACKET

<fillMethod> ::= FILL FILL_METHOD_NOPARAM LPAREN RPAREN <endline>
               | FILL FILL_METHOD_1PARAM LPAREN <parameter> RPAREN <endline>

<constraintPart> ::= CONSTRAINT <constr> <moreConstr> <endline>
<moreConstr> ::= <constr> <moreConstr> | <empty>
<constr> ::= CONSTR_NOPARAM
           | CONSTR_1PARAM LPAREN NUMBER RPAREN

<parameters> ::= <parameters> <1parameter> | <empty>
<1parameter> ::= PATH | REGEX | NUMBER | IDENTIFIER
               | IDENTIFIER COLON IDENTIFIER

<endline> ::= EOL <extraEndline>
<extraEndline> ::= EOL <extraEndline> | <empty>
<empty> ::=
```

3.2.2 Příklady konstrukcí jazyka pro konkrétní SQL tabulky

Mějme databázi D a v ní dvě tabulky `product_types` a `products`:

```
CREATE TABLE product_types (
    type_id serial PRIMARY KEY,
    name text,
    price numeric
);
CREATE TABLE products (
    product_id serial PRIMARY KEY,
    type_id integer REFERENCES product_types (type_id)
);
```

Způsob plnění databáze D je možné navrženým jazykem popsat např. následovně:

```
TABLE:product_types(100)                    //100 insertů do tabulky
    ::type_id
        TYPE serial
        FILL fm_basic()                     //základní metoda
        CONSTRAINT primary_key(0)
    ::name
        TYPE text
        FILL fm_textbank(./products.txt)    //textová banka products.txt
    ::price
        TYPE numeric
        FILL fm_basic

TABLE:products(5000)
    ::product_id
        TYPE serial
        FILL fm_basic()
        CONSTRAINT primary_key(0)
    ::type_id
        TYPE integer
        FILL fm_reference(product_types:type_id)
        CONSTRAINT foreign_key
```

Významem jednotlivých konstrukcí se zabývá kapitola 6.

3.3 Návrh nástroje pro generovní dat na základě popisu plnění v doménově specifickém jazyce

Druhý nástroj (pracovně označovaný symbolickým „Seeder“ - sázeč) bude na vstupu přijímat mezisoubor v doménove specifickém jazyce, na jehož základě bude generovat vhodné dotazy typu `INSERT` pro naplnění databáze. Seeder musí obsahovat parser doménově specifického jazyka aby byl schopen mezisoubor vhodně interpretovat při generování hodnot.

Podobně jako Harvester bude muset získané informace uložit do vlastních struktur, ze kterých bude následně čerpat generátor hodnot.

Nástroj by měl uživateli nabízet sadu variant plnění, jejichž použití bude možné specifikovat vhodnou úpravou mezisouboru.

3.3.1 Koncepce stěžejních metod pro generování hodnot

Informaci o způsobu plnění jednotlivých atributů Seeder získá analýzou mezisouboru. Metody, které nástroj bude poskytovat, jsou následující:

Základní plnění

Metoda aplikovatelná na všechny podporované datové typy. Pro každý z typů obsahuje algoritmus pro vygenerování validní hodnoty. Generovaná data budou obecná a uživatel nebude mít možnost do jejich vlastností zasáhnout (vyjma použitých integritních omezení).

Plnění na základě regulárních výrazů

Nejflexibilnější způsob plnění z pohledu zásahu uživatele. Uživatel bude mít při editaci mezisouboru možnost zadat vzor, podle kterého se bude obsah generovat. Toto je mu umožněno pomocí regulárních výrazů. Jednoduše tak bude možné dosáhnout naplnění sloupce například e-mailovými adresami.

Plnění s použitím hodnot dodaných v textových bankách

Tento způsob plnění bude vhodný především pro atributy obsahujících omezený počet opakujících se hodnot. Generátor bude mít k dispozici tzv. textové banky, jednoduché textové soubory, kde každý řádek bude považován za jednu možnou hodnotu k naplnění. Z těchto hodnot se bude při plnění náhodně vybírat.

Seeder bude mít k dispozici několik základních textových bank, uživatelská přívětivost této metody však bude založena primárně na možnosti dodání vlastních souborů a volby jejich použití v mezisouboru.

3.3.2 Kontrola integrity mezisouboru

Protože mezisoubor si uživatel může upravit/vytvořit sám a není tedy nutně produktem analýzy nástroje Harvester, je potřeba, aby Seeder prováděl vlastní sémantickou kontrolu. Tato by měla zahrnovat např. ověření kompatibility datových typů se zvolenou plnicí metodou nebo kontolu validity kombinací integritních omezení.

3.3.3 Výstup nástroje

Výstupem nástroje Seeder bude množina dotazů `INSERT` pro naplnění tabulek popsaných v mezisouboru pomocí DSJ.

Kapitola 4

Harvester – Implementace nástroje pro analýzu zdrojového dumpu

První nástroj byl implementován jako částečný parser dumpů generovaných PostgreSQL. Označení „částečný“ vyplývá ze skutečnosti, že Harvester ke své činnosti potřebuje jen dílčí informace v dumpech obsažené. Jedná se především o dotazy typu CREATE TABLE, ALTER SEQUENCE a některé dotazy typu ALTER TABLE. Jejich význam při získávání potřebných informací o zdrojové databázi bude dále popsán v kapitole 4.2.

Aby zabudovaný parser nástroje Harvester získal na vstupu pouze tyto vybrané dotazy, prochází dump nejprve přes tzv. preparser, který obstarává odfiltrování nepotřebných částí. Během následného parsování dochází k internímu vybudování struktury databáze pomocí instancí vhodně navržených tříd Table a Attribute. V poslední fázi jsou takto získaná data použita k vygenerování mezisouboru v DSJ. Tento soubor je okamžitě možné použít na vstupu druhého nástroje. Je však potřeba brát v potaz skutečnost, že se jedná o nejprimitivnější popis plnění všech tabulek, se stejným počtem dotazů `INSERT` pro každou z nich. Pokud si uživatel přeje hodnoty atributů blíže specifikovat, změnit počet generovaných dotazů, popř. plnit pouze fragment databáze, vlastní zásah do souboru je nutný.

Účelem nástroje je usnadnit uživateli proces plnění tím, že předgeneruje jakýsi mustr v doménově specifickém jazyce, který si následně uživatel může pouze upravovat k vlastní spokojenosti. Použití nástroje však není nezbytné, mezisoubor popisující plnění databáze si uživatel může vytvořit sám.

Hlavními moduly nástroje Harvester jsou:

harvester.py – Řídicí modul nástroje. Zajišťuje zpracování parametrů a postupné volání procedur. Obsahuje preparser.

lex_parse.py – Parser vybraných dotazů PostgreSQL. Obstarává lexikální a syntaktickou analýzu. Plní instance tříd Table a Attribute.

dsl_gen.py – Generátor konstrukcí v doménove specifickém jazyce. Na základě dodaných dat vytváří základní popis pro budoucí plnění databáze.

4.1 Parametry nástroje

Nástroj lze spustit s těmito přepínači:

Povinné

src	Cesta ke zdrojovému databázovému dumpu.

Volitelné

`-c N`, `--count N`	Počet generovaných plnění (N) pro každou tabulku.
`-s`, `--stdout`	Mezisoubor tištěn na standardní výstup.
`-f`, `--file`	Mezisoubor uložen do domovského adresáře jako dsl.txt.
`-h --help`	Tisk nápovědy.

Není-li specifikován počet plnění pomocí přepínače `-c`, bude použita přednastavená hodnota 10.

Přepínače `-s` a `-f` jsou výlučně volitelné. Není-li zadán žádný, automaticky se uvažuje volba `-s`, tisk na standardní výstup.

4.2 PostgreSQL dump a preparser

Vstup nástroje tvoří dump vygenerovaný z PostgreSQL databáze, která má být naplněna. Primárním účelem existence dumpu je zálohování databáze pomocí textového souboru. Opětovné předložení tohoto souboru databázovému serveru vytvoří přesnou kopii původní databáze včetně stavu, ve kterém se při tvorbě dumpu nacházela [10]. Je tedy nasnadě, že databázový dump zaznamenává do detailu všechny aspekty, které jsou pro tuto akci nezbytné, včetně samotných vložených dat.

Harvester potřebuje pro svou činnost pouze informace týkající se struktury relací, které mají být naplněny. Jedná se o jejich názvy a popis atributů, ze kterých se skládají. Pro každý atribut je nutné zjistit název, datový typ s případnými parametry, základní hodnotu, je-li dána, a informace o všech omezeních, která na něj byla použita.

V dumpu PostgreSQL verze 9.2.5 lze tyto údaje nalézt v dotazech trojího typu, jak je ukázáno na následujícím příkladu.

Př. 1 Mějme databázi obsahující pouze dvě tabulky `student` a `grp`.

```
CREATE TABLE student(
id serial NOT NULL PRIMARY KEY,
name varchar(64) DEFAULT '' NOT NULL,
specification char(3) NOT NULL,
year integer NOT NULL,
login char(8) DEFAULT '' not null,
grp_no integer NOT NULL REFERENCES grp(number)
);

CREATE TABLE grp(
number serial NOT NULL UNIQUE);
```

Pro tuto databázi jsme vygenerovali dump d. Fragmenty dumpu d nutné pro rekonstrukci tabulky `student` do podoby, ze které Harvester získá všechna potřebná data, jsou následující:

```
CREATE TABLE student (
    id integer NOT NULL,
    name character varying(64) DEFAULT ''::character varying NOT NULL,
    specification character(3) NOT NULL,
    year integer NOT NULL,
    login character(8) DEFAULT ''::bpchar NOT NULL,
    grp_no integer NOT NULL
);

...

ALTER SEQUENCE student_id_seq OWNED BY student.id;

...

ALTER TABLE ONLY student
    ADD CONSTRAINT student_pkey PRIMARY KEY (id);

...

ALTER TABLE ONLY student
    ADD CONSTRAINT student_grp_fkey FOREIGN KEY (grp_no)
                                            REFERENCES grp(number);
```

Jak je patrné na uvedeném příkladu, stěžejním dotazem pro získání kostry tabulky zůstává `CREATE TABLE`. Avšak omezení typu `PRIMARY/FOREIGN KEY` a také `UNIQUE`(není uvedeno v příkladu) jsou k atributům přiřazeny až následně pomocí dotazů `ALTER TABLE`. Dotaz `ALTER SEQUENCE` je jediným vodítkem, které nás informuje o skutečnosti, že atribut má v databázi přednastavené vlastní plnění. Často se jedná o atribut datového typu `int`/`bigint`, který byl zadán jako `serial`/`bigserial`.

Preparser zabudovaný do nástroje Harvester má za úkol tyto dotazy oddělit od ostatních. Nejedná se o parser v pravém slova smyslu – postrádá lexikální a syntaktický analyzátor, protože v tomto případě by takový postup vyžadoval schopnost zparsovat celý dump. Parser pro kompletní PostgreSQL však není předmětem této bakalářské práce.

Preparser namísto toho využívá vyhledávání přítomnosti klíčových slov v jednotlivých dotazech, v tomto případě tedy `CREATE TABLE`, `ALTER TABLE ONLY + KEY` nebo `ALTER SEQUENCE`. Tyto kombinace slov jednoznačně určují požadované dotazy, které jsou postupně ukládány do dočasného souboru a takto dále předány parseru.

4.3 Parser předfiltrovaných dotazů

Vlastní zabudovaný parser nástroje je schopen zpracovat pouze vybrané dotazy ze zdrojového dumpu. Ačkoliv databáze PostgreSQL podporuje na vstupu více variant dotazu téhož významu, výstup v podobě dumpu používá stále stejných konstrukcí. Díky této skutečnosti je parser jednodušší, souvisí s ní ale i jeho hlavní slabina: přísná závislost na konkrétní syntaxi přijímaných dotazů.

4.3.1 Modul PLY

Pro implementaci parseru byl použit **PLY** (Python Lex–Yacc). Jedná se o nástroj generující parsery na základě uživatelem dodané gramatiky a tokenů. **PLY** se skládá ze dvou hlavních modulů:

1. Generování lexikálního analyzátoru zajišťuje modul **lex.py**. Umožňuje uživateli specifikovat podporované tokeny pomocí regulárních výrazů (případně funkcí pro jejich další zpracování) nebo slovníku rezervovaných slov.

   Lexikální analyzátor generovaný pro potřeby nástroje Harvester ve velkém využívá především možnosti zadání rezervovaných slov. Kromě základních slov identifikujících typ dotazu (`CREATE`, `CREATE`, `TABLE` atp.) pokrývá také značnou část datových typů podporovaných PostgreSQL. Mezi nejdůležitější tokeny zadané regulárním výrazem patří především `IDENTIFIER` a `NUMBER`, tokeny zahrnující názvy relací/atributů a parametry datových typů.

2. Modul **yacc.py** má v režii tvorbu syntaktického analyzátoru. Uživatel zadává pravidla z gramatiky parsovaného jazyka formou pythonovských funkcí. Samotné pravidlo je ve formě BNF napsáno v docstringu funkce, prováděné operace jsou dále specifikovány v jejím těle, jak je ukázáno na příkladu:

   ```
   def p_expression_plus(p):
   'expression : expression PLUS term'
   #   ^            ^        ^    ^
   #  p[0]         p[1]     p[2] p[3]

   p[0] = p[1] + p[3]
   ```

   Jakmile dojde k přiřazení pravidla k množině tokenů a již redukovaných pravidel, je tato množina reprezentována objektem p. Tento objekt umožňuje jednoduchý přístup k hodnotám jednotlivých tokenů a následnou práci s nimi. Dokončením těla funkce dochází k redukci daného pravidla a postupu k pravidlu dalšímu.

   Yacc.py používá LR parser (konkrétně LALR variantu) vhodný k parsování deterministických bezkontextových gramatik [5].

4.3.2 Činnost parseru

Během parsování dochází k identifikaci tabulek a jejich atributů. K uchování získaných informací pro pozdější generování mezisouboru byly navrženy třídy Table a Attribute.

Objekt třídy Table uchovává dva důležité údaje: jméno tabulky a seznam objektů třídy Attribute reprezentujících atributy, které tuto tabulku tvoří.

Objekt třídy Attribute si nese informace o svém názvu, datovém typu a parametrech datového typu (jsou-li specifikovány). Obsahuje také velké množství příznaků pro jednotlivá omezení a další vlastnosti (např. jedná-li se o pole). V případě cizího klíče uchovává i název odkazovaného atributu a tabulky, ve které se tento nachází.

Narazí-li parser na pravidlo definující hlavičku dotazu pro tvorbu tabulky, vytváří novou instanci třídy Table, které přiřadí její název. Následuje parsování atributů tabulky, každý z nich je jako objekt třídy Attribute naplněn a připojen k seznamu atributů. Jakmile parser dojde na konec `CREATE TABLE` dotazu („;“), naplněný seznam je přiřazen objektu tabulky.

Tabulka sama je následně připojena k seznamu tabulek a také zařazena do slovníku 'název : objekt', který později umožňuje její snadnější vyhledání při dodatečných změnách vlastností atributů.

Dodatečnými změnami se rozumí přidání příznaků primárního/cizího klíče, omezení `unique` nebo aktualizace datového typu v závislosti na odhalení existence plnící sekvence.

Parser využívá a ukládá značnou část informací obsažených ve vybraných dotazech. Jedinou konstrukcí, kterou záměrně nechává bez povšimnutí, je notace přetypování `::datový typ`, která je v některých případech generována za hodnotu uvedenou v rámci omezení `DEFAULT`. Jedná se o zápis, který pouze informuje databázi, jak má nakládat se zmíněnou přednastavenou hodnotou, pro pozdější generování hodnot tedy není důležitý.

Při parsování primárních klíčů a omezení `UNIQUE` dochází k identifikaci, zda-li je unikátní pouze jediný atribut, nebo jedinečnost vzniká kombinací více atributů. Pakliže platí druhá varianta, všem zúčastněným atrbutům je přiřazeno unikátní číslo skupiny, do které spadají. Pokud je atribut unikátní samostatně, číslo jeho skupiny je vždy 0.

Další důležitou činností parseru je doplňování nespecifikovaných parametrů přednastavenými hodnotami. Například datové typy `bit`, `bit varying`, `character` či `character varying` lze obdržet bez parametrů specifikujících rozsah (popř. maximální rozsah). Pro účely generování mezisoboru a následné plnění je však tento stav nežádoucí, hranice je potřeba znát. Parser tedy v takovém případě doplňuje neexistující parametr defaultní hodnotou.

Výsledný seznam naplněných instancí třídy Table je návratovou hodnotou parseru, která je předána řídícímu modulu nástroje Harvester k dalšímu zpracování.

4.4 Generátor mezisouboru

Modul generátoru zajišťuje přepis informací uchovávaných v objektech tříd Table a Attribute do navrženého doménově specifického jazyka.

Ke generování dochází ve dvou vnořených cyklech. První cyklí nad seznamem objektů Table a generuje řádek s hlavičkou (název tabulky, počet plnění). Další cyklus obstarává popis každého atributu tak, jak to gramatika jazyka vyžaduje.

Generátor počítá s příchodem objektů, které už mají požadované kombinace uložených dat (např. datový typ a jeho parametry). K žádné vlastní kontrole validity zde nedochází, generátor pouze slepě převádí data na jazyk. Z tohoto důvodu vhodné kontroly a úpravy provádí už parser, který má k takové činnosti lepší prostředky (konkrétní pravidla).

Jedinou výjimkou je test kompatibility integritních omezení v rámci inicializace generátoru. Kontrola se zaměřuje na kombinace, ke kterým by bez externího zásahu do zdrojového dumpu nemělo dojít (např. `null` a `not null`).

Modul obsahuje slovník datových typů. Pokud se datový typ rozpoznaný parserem v tomto slovníku nachází, dojde k přepisu jeho názvu na název používaný ve výstupním jazyce (DSJ). Obvykle se jedná pouze o přepis slova v minuskách na jeho variantu ve verzálkách (`bit` – `BIT`), v některých případech přepis na kratší variantu (`integer` – `INT`). Varianty napsané verzálkami jsou datové typy, které rozpoznává Seeder. Kratší tvary datových typů byly zvoleny z důvodů úspory především při parsování mezisouboru, kterým se zabývá kapitola 5.2.

Nedojde-li k nalezení datového typu ve slovníku, slovo je generováno v minuskách, což značí, že se zatím nejedná o podporovaný datový typ pro generování dat. Obě varianty jsou uvedeny na následujcím příkladu:

```
TABLE:variable(10)
      ::name
              TYPE VARCHAR(128)
              FILL fm_basic()
              CONSTRAINT primary_key(0) default('')
      ::value
              TYPE bytea
                 FILL fm_basic()
```

`Varchar`, čili `character varying` je podporovaný datový typ. Generátor jej nalezl ve svém slovníku typů a přepsal na variantu ve verzálkách `VARCHAR`. Naopak datový typ `bytea` podporovaný není, zůstává minuskami.

Jediným nepodporovaným integritním omezením zůstává `CHECK`. V rámci plynulého parsování dumpu je sice i tento včetně parametrů uložen, při současné podobě práce však zatím není implementováno jeho dodržení. Generátor proto jeho zápis vynechává.

Finální výstupní mezisoubor v doménově specifickém jazyce generátor zapisuje na výstup zvolený přepínačem.

Kapitola 5

Seeder – Implementace nástroje pro generování dat

Principem činnosti druhého nástroje je tvorba plnících dotazů `INSERT` s příslušnými daty na základě popisu plnění dodaného v mezisouboru. Jedná se o významnější nástroj z dvojice Harvester, Seeder. Zatímco použití nástroje Harvester není pro finální výsledek nezbytné, Seeder tvoří jádro této práce.

Seeder na vstupu přijímá mezisoubor v doménově specifickém jazyce, který je následně zpracován zabudovaným parserem nástroje. Během parsování jsou data obdobně jako u Harvestrera vhodně ukládána do instancí dvojice tříd Table a Attribute. Takto jsou následně předána generátoru.

Generátor je nejdůležitějším modulem nástroje. Řídí správné rozdělení atributů, v jeho režii probíhá veškerá tvorba dat, kontroluje dodržení integritních omezení a uchovává data pro další plnění, je-li to nutné. Pod jeho správu spadají moduly plnících metod, jejichž použití si uživatel může zvolit úpravou mezisouboru.

Výsledkem činnosti nástroje je skript obsahující dotazy `INSERT`, jehož předložením zdrojové databázi dojde k požadovanému naplnění.

Hlavními moduly nástroje Seeder jsou:

seeder.py	–	Řídicí modul nástroje. Zajišťuje zpracování parametrů a postupné volání procedur.
lex_parse.py	–	Parser doménově specifického jazyka. Obstarává lexikální a syntaktickou analýzu. Plní instance tříd Table a Attribute.
data_gen.py	–	Generátor dotazů `INSERT` s vhodnými daty.

5.1 Parametry nástroje

Nástroj lze spustit s těmito přepínači:

Volitelné

`-s PATH`, `--source PATH` Vlastní umístění mezisouboru.

Není-li zadán přepínač `-s` s cestou k mezisouboru, Seeder se automaticky pokusí přistoupit k souboru dsl.txt v domovském adresáři uživatele.

5.2 Parser mezisouboru

Schopnost nástroje naplnit databázi daty se odvíjí od popisu v doménově specifickém jazyce, ve kterém je napsán vstupní mezisoubor. Aby byl nástroj schopen tento popis správně interpretovat, je nezbytné, aby obsahoval parser navrženého doménově specifického jazyka.

Obdobně jako u nástroje Harvester byl i v tomto případě pro implementaci parseru použit nástroj **PLY** (více v 4.3.1). Zde jsou ovšem narozdíl od parseru rozebíraného v kapitole 4.3.2 častěji používány i obecné tokeny (`IDENTIFIER`, `NUMBER`, `REGEX` atp.). Vyplývá to z podstaty parsovaných konstrukcí, zde na vstupu očekáváme zásahy od uživatele (např. cesta k textové bance), které klíčovými slovy pokrýt nelze.

Také třídy Table a Attribute jsou v nástroji Seeder robustnější. Obsahují více příznaků, které při generování umožňují rychlejší identifikaci vhodných množin dat. Nejvýraznější rozšíření lze nalézt ve třídě Attribute. Zde se kromě příznaků jedná především o uchovávání informací o zvolených plnicích metodách a jejich případných argumentech. Attribute si také pro potřeby pozdějšího generování nese proměnné přichystané k uchovávání již použitých hodnot (více v kapitole 5.3).

V parseru (ani později během generování) nedochází ke kontrole kolizí jmen tabulek nebo atributů v rámci jedné tabulky. Je-li mezisoubor výstupem prvního nástroje na základě validního PostgreSQL dumpu, k této situaci by nemělo. Může se vyskytnout pouze po nevhodném zásahu uživatele do mezisouboru.

Výstupem parseru je seznam naplněných instancí třídy Table, který vrací řídicímu modulu nástroje Seeder.

5.3 Generátor dat

Činnost modulu generátoru dat lze rozdělit na dvě části. Nejdříve dochází k inicializaci, která se zaměřuje na uvedení dat v objektech Table a Attribute do stavu, který umožňuje správné naplnění.

Po inicializaci dochází k samotnému procesu generování, během kterého probíhá kontrola atributů a volba vhodného postupu při tvorbě dat k naplnění.

5.3.1 Inicializace

Při volání generátoru dat dochází nejprve k jeho inicializaci. Generátor si při ní nezávisle prochází všechny tabulky a jejich atributy. Podobně jako v nástroji Harvester zde dochází ke kontrole kompatibility integritních omezení, slovo inicializace však vychází především z obsluhy atributů, které jsou cizími klíči.

Nepostradatelnými informacemi při tvorbě cizího klíče jsou údaje o atributu, na který cizí klíč odkazuje, a tabulce, které tento vzdálený atribut náleží. V SQL stejně jako v doménově specifickém jazyce jsou uvedeny jejich názvy. Jako takové je také zabudovaný parser identifikuje a uloží. Pro práci generátoru je však nutné, aby atribut typu cizí klíč měl ve svých proměnných odkaz na celé objekty, ne pouze jejich jména.

Proto pro každý nalezený cizí klíč probíhá vyhledávání referovaného atributu. Nejprve se podle uloženého jména dohledá jeho mateřská tabulka a v seznamu jejích atributů poté samotný atribut. Odkazy na tyto objekty jsou následně přepsány proměnné cizího klíče, které dosud nesly pouze jejich jména. U vzdáleného atributu je nastaven příznak `fk_pointed`. Tímto dochází k inicializaci cizích klíčů pro plnění.

V této části také probíhá kontrola korespondence datových typů odkazujícího a odkazovaného atributu a obecně existence odkazované tabulky a atributu.

5.3.2 Proces plnění

Po úspěšné inicializaci se přistupuje k samotnému plnění. Jeho princip spočívá v opakovaném iterování nad každou tabulkou, která má být naplněna. Počet cyklů se rovná počtu naplnění stanovenému mezisouborem.

Pro každou iteraci dochází k volání funkce `get_values()` nad danou tabulkou. Výstupem této funkce je řetězec s jedním dotazem typu `INSERT` s patřičnými hodnotami, který je okamžitě zapsán na výstup. Z toho vyplývá, že data jsou generována postupně, po malých sadách hodnot, okamžitě jsou vypsána a nikam se neukládají (výjimky probrány dále).

Tento postup byl zvolen s ohledem na možné extrémy při využití nástroje. Ty vychází z předpokladu, že testované databáze mohou být rozsáhlé a počet generovaných dotazů `INSERT` velký. Pokud by generátor nejprve všechna data ukládal do struktur a z nich až následně bral při tvorbě dotazů, paměť by na takový objem nemusela stačit. Naopak se dá předpokládat, že pontenciální uživatel nebude v časové tísni, proto se při implementaci přiklonilo k variantě vyšší časové a nižší paměťové náročnosti.

V důsledku volby této varianty však většinu chyb spojených s plněním (např. nedostatek hodnot dodaných v textové bance nebo nedostatečná práva) nelze určit dříve než během samotného plnění. Výskyt chyby znamená ukončení činnosti nástroje a dosud vygenerovaná data tudíž nejsou kompletní. Toto je jeden z hlavních důvodů, proč Seeder nepodporuje přímé plnění databáze, kde by přes parametry získal přístupové údaje a plnění obstarával sám postupným vkládáním dotazů `INSERT`.

Protože tabulky mohou svými cizími klíči referovat na atributy jiných tabulek, je také nutné zajistit, že budou plněny ve správném pořadí, tj. závislá tabulka bude naplněna až poté, co budou k dispozici hodnoty, na které referuje její cizí klíč. Proto jsou tabulky při každém pokusu o plnění testovány na přítomnost cizího klíče. Pokud žádný neobsahují, jsou naplněny a následně je jim nastaven příznak `solved`. Jestliže tabulka obsahuje cizí klíče a minimálně jeden z nich odkazuje na tabulku, která nemá příznak `solved` nastavený, plnění závislé tabulky je odloženo.

Kontroly a následná plnění probíhají v cyklu, který je přerušen až ve chvíli, kdy mají všechny tabulky příznak `solved` pozitivní, čímž generátor končí svou činnost.

5.3.3 Princip generování dat

Funkce `get_values()` obstarává vygenerovaní jedné sady hodnot pro naplnění dané tabulky. Získané hodnoty ve formě řetězců konkatenuje návzájem a vytváří syntakticky správný `INSERT`.

U každého atributu dochází ke kontrole, jedná-li se o cizí klíč, pole nebo sériově plněný atribut. Pokud je test negativní, funkce nad daným atributem zavolá zvolenou plnicí metodu (více v 5.3.4).

Je-li atribut identifikován jako sériově plněný (tzn. v databázi pro něj existuje plnicí sekvence), na místo hodnoty k naplnění se připojuje řetězec „DEFAULT“, který při vkládání do databáze zajistí dosazení následující hodnoty z dané sekvence.

Pokud je pozitivní příznak cizího klíče, hodnota atributu se získává odlišným způsobem (kapitola 5.4), podobně to platí také pro pole (kapitola 5.5).

Jakmile generátor obdrží nejnovější hodnotu, dochází k dalším testům, které ověřují nastavení příznaků `primary_key` nebo `unique`. Je-li výsledek pozitivní, je nutné ověřit, že nová hodnota již nebyla při plnění tohoto atributu použita.

Atributy s příznaky unikátních hodnot (myšleno samostatně unikátních, skupiny řešeny v kapitole 5.6) jsou jeden z případů, kdy je každá použitá hodnota uložena do seznamu, který je součástí objektů třídy Attribute. Pokud se tedy kontrolovaná hodnota shoduje s některým již vloženým prvkem, není akceptována a generuje se znovu. Jestliže však hodnota v seznamu není, je do seznamu zařazena a použita k plnění.

Takto jsou hodnoty ukládány i u atributů, které mají nastavený příznak `fk_pointed`. Důvodem je, aby byly k dispozici ve chvíli, kdy dojde na plnění na nich závislých cizích klíčů.

Při práci s nástrojem Seeder je nutné uvažovat skutečnost, že generátor je schopen zaručit splnění integritních omezení `unique` a `primary_key` pouze za předpokladu, že plněná tabulka již nemá v databázi vložené řádky. Vyplývá to z podstaty tohoto nástroje, který je realizován jako nezávislý na připojení k databázi a nemá tedy jak zjistit, že daná hodnota již byla v databázi použita. Takové situace musí uživatel řešit sám dle uvážení. Lze použít vhodnou plnicí metodu, která kolizi v daném případě znemožní, nebo počítat s možným menším počtem vložených dat (provedení kolidujících dotazů `INSERT` databáze nepovolí).

5.3.4 Plnicí metody

Seeder byl naimplementován s podporou čtyř plnicích metod, které přijímá na vstupu v doménově specifickém jazyce. Jedná se o metody, které mají za úkol demonstrovat možnosti nástroje.

fm_basic()

Základní plnicí metoda. Datový typ je nástrojem Seeder podporován ve chvíli, kdy může být naplněn touto metodou. Harvester ji při generování mezisouboru automaticky přiřazuje všem atributům, výjimkou jsou pouze cizí klíče.

Úkolem metody je zajistit vygenerování hodnoty, kterou databáze pro daný datový typ přijme. Tato všestrannost je však zaplacena generováním hodnot bez valného smyslu. Jediná jistota, kterou metoda dokáže poskytnout, je použitelnost generovaných hodnot.

Je vhodná především pro plnění atributů, na jejichž hodnotách uživateli příliš nezáleží, pouze je potřebuje naplněné. Je to také nouzové řešení při plnění typů, na které se zatím žádná jiná metoda nevztahuje.

fm_regex('REGEX')

Jedna ze ztěžejních metod nástroje pro generování dat. Jejím cílem je poskytnout uživateli při plnění téměř neomezenou volnost. Umožňuje mu přizpůsobit generovaná data jeho představě, kterou specifikuje pomocí regulárního výrazu zadaného jako parametr REGEX.

PostgreSQL databáze u některých typů vyžaduje vkládat hodnotu uvedenou v jednoduchých uvozovkách, v několika případech je dokonce potřeba použít prefix specifikující vkládaný datový typ (např. `bit'11001101'`). Tyto formality za uživatele řeší samotná funkce, která na základě datového typu generovanou hodnotu správně ošetří.

Omezení metody `fm_regex()` tkví v nemožnosti uvést v samotném regulárním výrazu jednoduché uvozovky, protože znak uvozovky slouží parseru k rozpoznání začátku a konce

parametru s regulárním výrazem. Další omezení týkající se složitosti výrazu se pojí s omezeními použitého nástroje **exrex** [1].

`Fm_regex()` je metoda, u které neprobíhají žádné kontroly korespondence obdrženého výsledku s datovým typem plněného atributu. Za tu je plně zodpovědný uživatel.

fm_textbank(PATH)

Poslední z trojice nejdůležitějších plnicích metod nástroje Seeder. Jestliže `fm_regex()` má uživateli dát volnou ruku, tato metoda je zde pro situace, kdy uživatel zná konkrétní množinu hodnot, ze kterých by chtěl daný atribut plnit.

Metoda přijímá jako parametr PATH cestu k textovému souboru, kde každý řádek (kromě posledního) považuje za jednu hodnotu k naplnění. Mezi těmito pak při zavolání náhodně vybírá.

Metoda je omezena na textové datové typy (`CHAR`, `VARCHAR`, `TEXT`).

fm_reference(TABLE:ATTR)

Metoda používaná pro plnění cizích klíčů. Obecně se dá aplikovat na jakýkoliv atribut, který si uživatel přeje naplnit hodnotami použitými jinde – generátor se k takovému atributu bude chovat jako k cizímu klíči.

Samotná `fm_reference()` na rozdíl od dříve zmíněných metod nemá vlastní modul a vlastně neexistuje jako taková. Parseru slouží k identifikaci, že má atributu nastavit příznak a parametry cizího klíče. Funkce pro samotné plnění cizích klíčů je pak přímo součástí modulu generátoru pod názvem `get_foreign()`.

5.4 Zpracování cizích klíčů

Jak bylo zmíněno v kapitole 5.3.3, při inicializaci generátoru jsou atributy odkazované alespoň jedním cizím klíčem označeny příznakem `fk_pointed` a hodnoty do nich generováne se ukládají. Dojde-li pak na samotný cizí klíč, stačí už pouze z těchto uložených hodnot brát.

Jiný přístup je potřeba v situaci, kdy má cizí klíč zároveň nastaven i příznak `unique`. Pro všechny ostatní případy se tato situace zcela řeší v kořenové funkci `get_values()` spolu s ostatními kontrolami, v kombinaci s cizím klíčem je však vyžadován vlastní postup.

Při prvním zavolání funkce `get_foreign()` nad atributem dochází k rozlišení, jedná-li se o obyčejný cizí klíč, nebo zda je přítomen i příznak `unique`. Pokud platí první varianta, do speciální proměnné atributu se vloží odkaz na seznam hodnot, ze kterých se má brát (hodnoty uložené při plnění odkazovaného atributu). Při dalším plnění se pak vybírá náhodná hodnota z tohoto seznamu.

Pokud platí varianta s příznakem `unique`, seznam s hodnotami se do proměnné duplikuje a každá náhodně vybraná hodnota se z něj zároveň odstraní. Dojde-li k vyprázdnění seznamu, přestože generátor ještě nedokončil plnění klíče, dojde k výpsání chybového oznámení a ukončení činnosti nástroje.

Speciálním případem cizího klíče jsou předvyplněné tabulky v databázi. Je možné je použít, pokud si uživatel přeje naplnit pouze fragment databáze a některé plněné atributy odkazují na tabulky, které už jsou naplněné požadovanými hodnotami. Uživatel tyto tabulky

[1] Jedná se o Open Source (GNU AGPL) nástroj pro generování řetězců odpovídajících dodanému regulárnímu výrazu. Více v dokumentaci nástroje [1].

nemaže z mezisouboru, ale místo počtu naplnění napíše klíčové slovo „FILLED“. Takové tabulky generátor neplní, ale ví o jejich existenci a proto je může použít.

Pokud se pak při plnění cizího klíče narazí na atribut odkazující na takovou tabulku, je hodnota k jeho naplnění vygenerována jako dotaz `SELECT` s náhodným výběrem z daného sloupce odkazované tabulky. Obdobně se zachází také při plnění cizího klíče odkazujícího na sekvencí plněný atribut.

Při použití tohoto postupu Seeder nemůže žádným způsobem zaručit splnění integritního omezení `UNIQUE`.

5.5 Postup při plnění polí

Protože PostgreSQL umožňuje atribut zadat jako n-rozměrné pole daného datového typu, byl i Seeder naimplementován s podporou plnění takových atributů.

Je-li ve funkci `get_values()` nalezen pozitivní příznak pole, získání hodnoty pro naplnění tohoto atributu je předáno funkci `get_array()`.

Algoritmus použitý v této funkci je navržen tak, aby byl schopen naplnit pole o n rozměrech, kdy každý každý rozměr má velikost m_x, kde x je pořadové číslo rozměru. Princip algoritmu tkví v postupné, zpětné tvorbě n-tic.

Nejprve je vygenerován seznam požadovaného množství elementárních koncových hodnot. Jejich počet se získá vzájemným vynásobením velikostí všech rozměrů pole, pro pole $x[a][b][c]$ je tato hodnota rovna $a * b * c$. Tyto hodnoty jsou následně skládány do n-tic podle posledního rozměru, kde $n = c$ a opět ukládány jako seznam hodnot. Obdobně jsou tyto nové hodnoty použity pro vytvoření n-tic pro další rozměr, $n = b$. Takto se postupuje, dokud nezbývá pouze jedna, finální hodnota.

Generátor tvoří n-tice takovým způsobem, aby odpovídaly syntaxi PostgreSQL pro pole [10]. Poslední získaná hodnota je tedy vhodná k vložení do databáze po atribut pole daného rozměru a datového typu.

Závislost počtu iterací vnitřního cyklu na množství dimenzí a jejich velikosti lze zapsat následujícím vzorcem:

$$\sum_{i=1}^{n} m_0 * ... * m_i$$

kde n je počet dimenzí pole, $m_0 = 1$ a m_i je velikost daného rozměru pro $i \in < 1, n >$.

Postup je v generátoru implementován následovně:

```
old_list = val_list    #val list obsahuje vygenerované elementární hodnoty
    new_list = []

    for i in range(0,cnt):                #cnt obsahuje počet dimenzí
        j = cnt - (i+1)
        size = attr.array_dim_size[j]     #velikost poslední dimenze

        total = total / size

        for x in range(0,total):
            new_val = "{"
```

```
            for y in range(0,size):
                val = old_list.pop()
                new_val = new_val + val + delim

            new_val = new_val[:-1] + "}"
            new_list.append(new_val)

        old_list = new_list
        new_list = []

    new_val = "'" + old_list.pop() + "'"
```

Algoritmus se skládá ze tří vnořených cyklů, které zajišťují skládání elementárních hodnot dříve vygenerovaných do seznamu `val_list`.

Nejvíce zanořený cyklus vytváří n-tice z hodnot v `old_list`, kde n je velikost momentálně posledního rozměru. Konkatenuje je navzájem a k jejich oddělení používá oddělovač uložený v proměnné `delim`. Pro většinu datových typů se jedná o čárku, pro typ `BOX` je to středník [10]).

Prostřední cyklus pak zajišťuje vytvoření správného počtu n-tic, iteruje tolikrát, kolik je v současné chvíli součin velikostí všech dimenzí kromě dimenze poslední (tento součin = proměnná `total`).

Vnější cyklus má jako jediný pevně stanovený počet iterací, ten je roven počtu dimenzí pole. V každém cyklu proměnnou `total` dělí velikostí momentálně poslední dimenze, která je následně použita pro tvorbu n-tic. Tímto pole pro příští iteraci vždy o jeden rozměr zmenší.

Na konci každé iterace vnějšího cyklu dochází k nahrazení seznamu v `old_list` seznamem nově vygenerovaných n-tic `new_list`, aby se tyto mohly použít ke skládání v další iteraci. `New_list` je opět iniciován jako prázdný.

Po ukončení vnějšího cyklu seznam `old_list` obsahuje jedinou hodnotu, která vznikla postupným složením n-tic reprezentujících rozměry plněného pole. Tato hodnota je po konkatenaci s dvojitými uvozovkami vhodná k naplnění daného atributu (pole musí být vkládána ve dvojitých uvozovkách [10]).

5.6 Zajištění unikátních kombinací hodnot

SQL umožňuje, aby se omezení `UNIQUE` či `PRIMARY KEY` nevztahovala pouze na jediný atribut, ale také na set více atributů. Tyto atributy samy mohou nabývat opakujících se hodnot, ale jejich vzájemná kombinace musí být v tabulce unikátní.

Jak bylo popsáno v kapitole 4.3.2, rozlišení těchto dvou případů je v DSJ řešeno parametrem omezení `unique`/`primary_key`. Číslo 0 znamená samostatný atribut, číslo různé od nuly je identifikátorem skupiny atributů, které tvoří jednu unikátní kombinaci.

Při generování nového dotazu `INSERT` se skupinově unikátní hodnoty přidávají do dočasného seznamu. Ten je následně porovnáván s uloženými seznamy již použitých kombinací, které jsou uchovávány v proměnné `unique_values` instance třídy Table. Nebyla-li nová kombinace hodnot použita, je do tohoto seznamu zařazena a vygenerovaný dotaz může být použit. Pokud však dojde ke kolizi, proces vytvoření dotazu je nutné opakovat.

Tato situace s sebou nese možné problémy. Právě odhazované hodnoty totiž mohly být zaznamenány pro další použití. Pokud by zůstaly uloženy, mohlo by při nadcházejícím plnění

dojít k nekonzistenci. Toto se vztahuje na atributy s nastaveným příznakem `fk_pointed`, nebo atributy s číslem unikátní skupiny 0 (tj. samostatně unikátní).

Pro tyto případy byla naimplementována funkce `remove_last_saved()`, která projde všechny atributy tabulky a splňuje-li některý jednu z výše uvedených podmínek, z jeho seznamu uložených hodnot je odstraněna poslední vložená.

Seeder momentálně nepodporuje více jak jednu unikátní skupinu atributů na tabulku.

5.7 Nepodporované datové typy

Seeder v současné chvíli nepodporuje tyto datové typy:

- `bytea`
- `interval`
- `money`
- `tsquery`
- `tsvector`
- `txid_snapshot`
- `uuid`
- `xml`
- `line`[2]

Přesto je možné tyto typy naplnit, pokud uživatel v mezisouboru změní jejich datový typ na některý z podporovaných a jako plnicí metodu zvolí např. `fm_regex()` nebo `fm_textbank()`. Dodáním vhodných parametrů pak může docílit vygenerování požadovaných hodnot. Pokud ovšem vložení hodnoty vyžaduje přítomnost prefixu (jak bylo popsáno v 5.3.4), bude nutný přímý zásah do vygenerovaných dotazů `INSERT`.

[2]Nelze použít, v PostgeSQL není plně implementováno.

Kapitola 6

Práce s doménově specifickým jazykem

Tato kapitola se bude zabývat základním popisem konstrukcí doménově specifického jazyka a jejich použitím pro docílení požadovaného způsobu plnění.

DSJ byl navržen tak, aby umožnil uživateli co nejjednodušší specifikaci jeho požadavků na generování dat pro plnění. Aby mohlo být docíleno této vlastnosti, konstrukce jazyka reflektují schéma databáze (popř. fragmentu databáze), která má být plněna, konkrétně tabulek a jejich atributů.

Na pořadí uvedených tabulek nezáleží, pořadí atributů v rámci každé z nich je však nutné zachovat tak, aby odpovídalo jejich pořadí v plněné databázi. Toto je nutné z hlediska tvorby dotazů `INSERT`, které jsou generovány tak, že nespecifikují sloupce v pořadí, ve kterém vkládají hodnoty. Databáze tudíž očekává, že první vkládaná hodnota náleží prvnímu sloupci, druhá druhému atd. Prohození pořadí atributů by v drtivé většině případů znamenalo vkládání nekompatibilních hodnot a tudíž chybu.

Konstrukce ovlivňující generování hodnot budou podrobně probrány v následujících podkapitolách.

6.1 Nakládání s bílými znaky

V doménově specifickém jazyce hraje významnou roli pouze znak konce řádku. Jeho přítomnost je nutná na konci každé z konstrukcí uvedených v následujících podkapitolách, protože pro parser se jedná o důležitý ukončovací znak. Mimo tyto konstrukce se může odřádkování používat neomezeně.

Zbývající bílé znaky, konkrétně mezery a tabulátory, se mohou vyskytovat v libovolném množství, protože parser jejich přítomnost nebere v úvahu. Je tedy možné konstrukce v DSJ odsadit tak, aby byl výsledný mezisoubor pro uživatele co nejpřehlednější.

Doménově specifický jazyk nepodporuje žádnou formu komentářů.

Upozornění: Parser vyžaduje minimálně jeden prázdný řádek na konci mezisouboru. Jinak dochází k chybě, jejíž příčina je stěží dohledatelná.

6.2 Hlavička tabulky

Řádek s hlavičkou tabulky v DSJ může vypadat např. takto:

```
TABLE:example(100)
```

Důležité informace obsažené v tomto řádku jsou následující:

- V popisu plnění se začínáme věnovat nové tabulce.
- Názvev této tabulky je „example“
- Pro tuto tabulku má být vygenerováno 100 dotazů `INSERT`.

Názvy tabulek jsou při inicializaci generátoru kontrolovány na duplicity. Pokud se na kolizi narazí, nástroj je ukončen s chybovým hlášením.

Parametrem tabulky nemusí být nutně číslo, jak již bylo naznačeno v kapitole 5.4. Je-li tato tabulka již v databázi naplněná a uživatel si pouze přeje, aby Seeder o existenci této tabulky věděl, hlavičku modifikuje následovně:

```
TABLE:example(FILLED)
```

Tato informace umožňuje generátoru, aby mohl správně přistoupit k plnění cizích klíčů, které referují atribut této tabulky.

Zbývající konstrukce takovéto předvyplněné tabulky uživatel musí ponechat. Seeder je používá ke kontrole, zda-li daný atribut v tabulce existuje a je-li referovaný datový typ kompatibilní.

6.3 Hlavička atributu

Každý nový atribut je uvozen řádkem:

```
::attribute1
```

Informuje nás o existenci dalšího atributu a jeho názvu (v tomto případě „attribute1“.

Při práci s mezisouborem v DSJ, především při jeho vlastní tvorbě bez použití prvního nástroje, je nutné počítat se skutečností, že Seeder neprovádí kontrolu duplikátů mezi názvy atributů téže tabulky. Pokud uživatel zanese do souboru takovouto chybu, může to vyústit v nekonzistenci při plnění v závislosti na vlastnostech kolidujících atributů.

V případě referování cizím klíčem by byl například jako referovaný označen vždy první atribut v pořadí. Toto by mohlo vést ke kolizi datových typů, v lepším případě pouze k naplnění hodnotami z jiného sloupce, než bylo zamýšleno.

6.4 Datový typ

Po uvedení nového atributu vždy musí následovat řádek s datovým typem. Např.:

```
TYPE BIT(8)
```

nebo

```
TYPE INT
```

Datový typ musí být vždy uveden klíčovým slovem „TYPE“. Následuje samotný název typu a případné parametry.

Datové typy v DSJ používají pouze jedno označení (na rozdíl od SQL, kde je možné jeden typ uvést několika názvy). Je tomu tak především kvůli zjednodušení práce jak parseru mezisouboru, tak samotnému uživateli.

Názvy byly voleny tak, aby obsahovaly pouze jedno slovo[1], ideálně nejkratší variantu z oficiálních SQL názvů. Výjimku tvoří například datový typ `double precision`, jehož jedinou alternativní SQL variantou je `float8`. Za účelem co nejintuitivnější manipulace uživatele s mezisouborem je proto tento typ v DSJ znám jako `DOUBLE`.

Výskyt parametrů u konkrétního datového typu koresponduje s jeho SQL vzorem. Jedinou výjimkou jsou datové typy s nepovinnými parametry, tyto jsou v doménově specifickém jazyce povinné.

Důvodem je povaha zmíněných parametrů. SQL jejich absenci nahradí přednastavenou hodnotou (parametr `scale` u datového typu `numeric`, fixní délka u typu `char`), nebo, a to je obvyklejší, považuje vlastnost atributu, kterou parametr stanovuje, za neomezenou (např. parametry omezující rozsah datových typů `varchar`, `varbit`). Zvláštním případem jsou n-rozměrná pole, jejichž dimenze jsou v PostgreSQL momentálně reprezentovány jako nekonečné, ať už uživatel při jejich tvorbě velikost explicitně specifikoval, nebo ne [10].

Je zřejmé, že neurčitá informace o potenciálně nekonečně velkém atributu je pro práci nástroje Seeder nepotřebná, ne-li přímo nežádoucí. Nástroj byl navržen a implementován tak, aby uživateli usnadnil práci při tvorbě testovacích dat, není jeho úkolem pokoušet hranice možností plněné databáze. Proto parser nástroje přítomnost těchto parametrů vyžaduje a při generování se jejich významem řídí.

Harvester při generování jazyka tyto parametry automaticky plní vlastními přednastavenými hodnotami.

Podporované datové typy SQL a k nim korespondující typy v DSJ jsou zobrazeny v tabulce 6.1.

6.5 Specifikace plnicí metody

Řádek popisující plnicí metodu, která má být použita, musí následovat bezprostředně po uvedení datového typu. Může vypadat takto:

```
FILL fm_basic()
```

Klíčové slovo „FILL“ řádek uvozuje, následuje zvolená plnicí metoda. Princip metod byl blíže popsán v kapitole 5.3.4. Tato podkapitola se zaměří především na popis algoritmů použitých pro plnění nejvýznamnějších datových typů pomocí plnicí metody `fm_basic()`.

[1] Modul **PLY** plně podporuje pouze jednoduchá klíčová slova (bez mezer), pro víceslovné datové typy by tedy bylo nutné vytvořit speciální pravidla v gramatice DSJ, což by vedlo k zbytečnému zvýšení její složitosti.

SQL	DSJ	Parametry
bigint	BIGINT	
bigserial	BIGSERIAL	
bit	BIT	(n)
bit varying	VARBIT	(n)
boolean	BOOL	
box	BOX	
character varying	VARCHAR	(n)
character	CHAR	(n)
cidr	CIDR	
circle	CIRCLE	
date	DATE	
double precision	DOUBLE	
inet	INET	
integer	INT	
lseg	LSEG	
macaddr	MACADDR	
numeric	NUMERIC	(p,s)
path	PATH	
point	POINT	
polygon	POLYGON	
real	REAL	
serial	SERIAL	
smallint	SMALLINT	
time	TIME	(p,+/-TMZ)
timestamp	TIMESTAMP	(p,+/-TMZ)
text	TEXT	

Tabulka 6.1: Datové typy podporované v DSJ a jejich parametry

6.5.1 Metoda fm_basic() - praktické použití

Jak bylo vysvětleno dříve, především v kapitole 5.3.4, datový typ je nástrojem Seeder podporován ve chvíli, kdy je pro něj v této funkci implementováno plnění. Jak jsou tvořeny hodnoty nejpoužívanějších datových typů bude popsáno v následujících odstavcích.

Character varying - VARCHAR

Datový typ `VARCHAR` přijímá v parametru maximální povolenou délku. Pro potřeby plnění pomocí `fm_basic()` je mu dále stanovena minimální možná hranice 2. Je-li zadané maximum rovno 1, pak i minimum je nastaveno na 1.

Při generování je používán nástroj **exrex**, jemuž je předložen vhodný regulární výraz. Výsledná hodnota je pak ořezána na náhodně zvolenou délku z intervalu $< min, max >$, kde min a max značí výše zmíněnou minimální a maximální hranici.

Použitý regulární výraz vypadá následovně:

```
[BCDFGHJKLMNPQRSTVWXZ][aeiouy]([bcdfghjklmnpqrstvwxz][aeiouy]
                                                    [bcdfghjklmnpqrstvwxz]?)+
```

Záměrem je generovat řetězce připomínající slova. Výraz popisuje řetězce složené ze slabik, pro které vždy platí, že obsahují souhlásku a samohlásku (případně dodatečnou souhlásku) v tomto pořadí. Snaha napodobit slova je založena na předpokladu, že tento datový typ s proměnlivou délkou je často používán pro jména a názvy.

Příklad hodnot generovaných pro `VARCHAR(8)`:

Sofab

Tapinhu

Zowezin

Dyx

Mehyfuve

Hisab

Character - CHAR

U datového typu `CHAR` byl použit podobný postup, jako v předchozím případě, tedy generování na základě regulárního výrazu. Zde je ovšem regulární výraz obecnější:

```
[a-zA-Z0-9_]+
```

Jeho volba vycházela z předpokladu, že datový typ fixní délky může být častěji použit také pro kódy apod.

Příklad hodnot generovaných pro `CHAR(5)`:

RlM93

A8Q_2

fTF5d

b6NBo

ubZxT

B7Rv9

Text - TEXT

Pro generování hodnot datového typu `TEXT` byla vytvořena speciální textová banka obsahující celé věty. Z toho vyplývá, že metoda `fm_basic()` u tohoto typu při plnění počítá s větším množstvím textu. Pokud uživatel použil datový typ `TEXT` pro atribut uchovávající například jména, může datový typ pro účely plnění změnit například na `VARCHAR`.

Apostrofy ve větách (jedná se o věty v anglickém jazyce) byly z důvodu jejich správné interpretace při vkládání zdvojeny. Jedná se o způsob, jakým databáze tento znak zbavuje speciálního významu. Tím je obvykle uvození vkládané hodnoty.

Příklad hodnot generovaných pro `TEXT`:

> "Mike, you"re the proud father of a fine strapping boy."
>
> End of the June Days Uprising in Paris.
>
> The waitress arrived with the coffee and placed it on the table.
>
> "That"s easy"says the other one.
>
> How do you count a herd of cattle?
>
> They each bought a pint of Guinness.

Integer - INT a varianty BIGINT, SMALLINT

Pro plnění datových typů `INT`, `BIGINT` a `SMALLINT` se používá stejná sub-metoda, pouze jsou před jejím voláním na základě konkrétního datového typu nastaveny proměnné `CURRENT_MAX` a `CURRENT_MIN`, které odpovídají hraničním hodnotám daného typu v PostgreSQL [10]. Výsledná hodnota je ošetřena, aby spadala do intervalu tvořeného těmito hodnotami.

Generované číslo je vždy kladné. Předpokládá se, že kladná celá čísla jsou používána častěji. Obecně jsou pro typ `INT` generovaná v řádech statisíců až milionů, pro menší hodnoty je vhodné použít typ `SMALLINT`.

Příklad hodnot generovaných pro `SMALLINT`:

> 5429
>
> 3
>
> 16119
>
> 4325
>
> 12
>
> 27652

Real/Double precision - REAL a DOUBLE

Podobně jako v případě typu `INT` je také generování hodnot pro `REAL` a `DOUBLE` řešeno v téže sub-metodě, rozdíl opět tkví v použitých hraničních hodnotách. (Hodnoty zvoleny na základě Standardu IEEE754 [7], kterým se PostgreSQL řídí [10].)

Pro každý typ dojde nejprve k náhodnému vygenerování exponentu v povoleném rozmezí, následně je vytvořena i mantisa. Pro mantisu platí, že pro potřeby nástroje Seeder je vždy generována ve tvaru:

$$n.xyz$$

kde n je je jedna cifra v celočíselné části a x,y,z jsou cifry v desetinné části.

Následně dochází ke konkatenaci řetězců s mantisou a exponentem do finálního tvaru, který je poté převeden na pythonovskou reprezentaci typu `float`. Touto operací může dojít k interní úpravě tvaru z formátu mantisa-exponent na celé či desetinné číslo v závislosti na vygenerované kombinaci.

Příklad hodnot generovaných pro `REAL`:

2.606e+12

112000000.0

8.899e-34

5954.0

7.488e+35

6631000000.0

Časové typy - TIME, DATE a TIMESTAMP

Reprezentace data, času a časová razítka obecně jsou další typy, u kterých se předpokládá časté použití. PostgreSQL podporuje rozmanitý způsob jejich reprezentace, pro potřeby nástroje Seeder však byla pokaždé zvolena pouze jedna, nejintuitivnější varianta.

Datový typ `DATE` je tedy generován ve formátu `yyyy-mm-dd`, pro typ `TIME` je použit formát `hh:mm:ss` a typ `TIMESTAMP` je z předešlých dvou typů skládán.

Generovaný letopočet pro datový typ `DATE` je náhodně vybírán z intervalu $< 1990, 2013 >$. Interval byl zvolen na základě předpokladu, že častější použití letopočtu se vztahuje k zachování událostí, které se již staly (rok registrace, rok narození atd.).

Typy `TIME` a `TIMESTAMP` mají v SQL možnost zadání dvou parametrů. Jedná se o parametr `p`, který udává počet desetinných míst u vteřinové části[2], dále pak parametr `with/without time zone`, který stanovuje nutnost přístomnosti časové zóny.

Jak bylo popsáno v kapitole 6.4, Seeder všechny původně nepovinné parametry vyžaduje, v tomto případě tedy požadovaný formát v DSJ pro tyto datové typy vypadá následovně:

```
TYPE TIME(0, +TMZ)
TYPE TIMESTAMP(2, -TMZ)
```

[2]Podporovaný rozsah hodnot parametru p se pohybuje v intervalu $< 0, 6 >$.

Parametr označující přítomnost časové zóny byl pro zjednodušení zkrácen na tvary `+TMZ` (s časovou zónou), `-TMZ`(bez časové zóny).

Pro účely generování bylo vytvořeno pole obsahující výběr zkratek používaných pro identifikaci konkrétních časových zón. Z tohoto pole je při přítomnosti parametru `+TMZ` náhodně vybíráno.

Příklad hodnot generovaných pro `TIMESTAMP(0,+TMZ)`:

2011-04-01 16:05:15 EET

2009-07-15 00:30:23 PST

1999-03-29 10:55:30 AST

1997-04-14 03:10:39 CST

1994-07-30 04:27:37 GMT

2011-07-15 09:31:34 EST

6.6 Specifikace integritních omezení

Po uvedení povinných řádků specifikujících datový typ a plnicí metodu může následovat nepovinný řádek definující integritní omezení, která by měl Seeder při generování hodnot vzít v potaz. Tento řádek může vypadat následovně:

```
CONSTRAINT null(30)
CONSTRAINT foreign_key default('')
```

Řádek je opět uvozen klíčovým slovem, v tomto případě „CONSTRAINT“, následuje samotné integritní omezení. V nástroji Seeder jsou z důvodu snazšího parsování dvouslovná omezení spojena podtržítkem. Obecně jsou pak psána malými písmeny[3].

Integritiní omezení je možné i řetězit, jak je patrné na výše uvedeném příkladu. Seeder však při nalezení nekompatibilních kombinací (např. `null` a `not_null`) končí svou činnost s chybovým hlášením.

Podporovaná integritní omezení jsou následující:

- `unique(id)`
- `primary_key(id)`
- `foreign_key`
- `null(n)`
- `not_null`
- `default(n)`

[3]Pozor, parser nástroje Seeder je case-sensitive.

6.6.1 Omezení unique a primary_key

Kapitola 5.6 se zabývala významem parametru integritních omezení `unique` a `primary_key`. Harvester tyto parametry při tvorbě mezisouboru vhodně plní. Pokud však mezisoubor tvoří sám uživatel, je nutné, aby měl na paměti, že samostatně unikátní atributy obsahují v parametru vždy číslo 0. Naopak, jedná-li se o skupinu více atributů, které tvoří unikátní kombinaci, u všech těchto atributů je nutné uvést totéž číslo různé od 0.

V kapitole 5.6 bylo také probráno, že Seeder momentálně podporuje pouze jednu unikátní skupinu na tabulku. Je tedy možné unikátní skupiny ve všech tabulkách označovat stejným číslem různým od 0, protože princip jejich zpracování neumožňuje kolizi mezi skupinami z různých tabulek. Přesto je však doporučeno pro přehlednost dodržet unikátnost označení skupiny napříč všemi tabulkami, podobně jako to dělá Harvester.

6.6.2 Omezení foreign_key

Omezení `foreign_key` má spíše symbolický význam. Je vhodné jej vypsat za účelem větší přehlednosti mezisouboru. Příznak `foreign_key` je totiž u atributu nastaven pouze, pokud je zadána plnicí metoda `fm_reference()`, na přítomnosti vypsaného omezení `foreign_key` nezáleží. Z toho vyplývá, že toto omezení je možné napsat u kteréhokoliv atributu a nikdy nedojde k vypsání kolizní chyby, protože kromě povolené přítomnosti tohoto zápisu tento sám o sobě není brán v úvahu.

6.6.3 Omezení null a not_null

Podobně jako u omezení `unique` a `primary_key` byl pro potřeby generování přidán parametr i k omezení `null`. V tomto případě však číslo v závorce představuje procentuální šanci, že bude daný atribut naplněn hodnotou „NULL“.

Seeder ze své podstaty naplní vždy všechny atributy platnými hodnotami, dá se tedy uvažovat o nezávazném přednastaveném omezení `not_null`. Pokud si uživatel přeje, aby sloupec tabulky z části hodnotu „NULL“ obsahoval, může si její výskyt při plnění přizpůsobit právě pomocí parametru omezení `null`. Harvester při jeho nalezení automaticky předvyplňuje parametr hodnotou 20, toto omezení se však v dumpech nevyskytuje, protože se považuje za implicitní. Do mezisouboru je tedy nutné ho přidat ručně.

Obsahuje-li parametr číslo menší než 1, hodnota „NULL“ není k plnění použita nikdy, naopak je-li číslo v závorce větší nebo rovno hodnotě 100, atribut bude plněn pouze „NULL“.

Jak bylo zmíněno výše, Seeder standardně ke všem hodnotám přistupuje, jako by měly interně nastaveno omezení `not_null`. Jeho explicitní vypsání je však nutné z důvodu správné kontroly kolizí s jinými integritními omezeními[4].

6.6.4 Přednastavená hodnota default

Ačkliv se nejedná o integritní omezení v pravém slova smyslu, byla hodnota `default` do těchto v rámci doménově specifického jazyka zahrnuta.

Funguje na stejném principu jako omezení `null`. Na základě hodnoty čísla dodaného jako parametr jsou generovány buď platné hodnoty pro plnění, nebo je na jejich místo dosazen řetězec „DEFAULT“. Ten je pro databázi signálem, že má na jeho místo dosadit přednastavenou hodnotu.

[4]Za předpokladu, že plněný atribut má v databázi skutečně nastaveno omezení NOT NULL. Pokud tento zadaný není, není důvod ho uvádět ani v DSJ.

Omezení `null` a `default` lze v DSJ kombinovat. Při zadávání parametrů je však potřeba uvažovat skutečnost, že nejprve probíhá výpočet procentuální šance pro `default` a následně nezávisle na předchozí činnosti dojde k téže akci pro `null`. Omezení `null` tedy může přepsat hodnotu, která již byla stanovena jako „DEFAULT“.

Kapitola 7

Testování implementovaných nástrojů

Činnosti nástrojů Harvester i Seeder byly vyzkoušeny na několika testovacích databázích, s jejichž použitím bylo zkoumáno několik vlastností těchto nástrojů.

Testované vlastnosti byly následující:

- Schopnost nástroje Harvester zpracovat zdrojový dump a vygenerovat syntakticky správný mezisoubor.
- Schopnost nástroje Seeder zparsovat mezisoubor v DSJ a získaná data správně uložit do struktur.
- Generování validních hodnot pro všechny podporované datové typy pomocí `fm_basic()`.
- Funkčnost zbývajících plnicích metod.
- Správné plnění atributů na základě uvedených integritních omezení.
- Použitelnost nástrojů při automatizaci.

Použité databáze byly z větší části vytvořeny za účelem ověření konkrétní vlastnosti. Pro ověření použitelnosti nástrojů v reálném nasazení byla použita databáze systému pro správu obsahu Drupal.

7.1 Testování správné činnosti nástroje pro analýzu zdrojového dumpu

Pro ověření správné činnosti prvního nástroje byla více než malé pokusné databáze vhodná databáze systému Drupal. Důvodem je její komplexnost co se týče použitých SQL konstrukcí (indexy, funkce apod.) a také množství již vložených dat. Přítomnost těchto je pro testování nástroje Harvester vhodná z hlediska kontroly funkčnosti preparseru, jehož úkolem je vyfiltrovat všechna data nerelevantní pro činnost nástroje.

7.1.1 Práce preparseru na dumpu systému Drupal

Dump vygenerovaný pro databázi čerstvě iniciovaného Drupalu obsahuje bezmála 10000 řádků. Dočasný soubor vytvořený jako výstup preparseru už se skládá pouze z 900 řádků. Konstrukce důležité pro práci nástroje Harvester tedy v dumpu rozsáhlejší databáze tvoří méně než 10%.

Dočasný soubor obsahuje pouze konstrukce popsané v kapitole 4.2. Ty se týkají 74 tabulek a jejich atributů. Přes 70% souboru tvoří dotazy `CREATE TABLE`, zbývající část se skládá převážně z dotazů `ALTER TABLE`. Dotazy `CREATE SEQUENCE` v tomto případě v celém souboru zabírají pouze tři řádky.

7.1.2 Generování mezisouboru pro plnění databáze systému Drupal

Dočasný soubor s předfiltrovanými dotazy popsaný v předchozí kapitole byl dále předložen parseru a generátoru doménově specifického jazyka. Výsledný mezisoubor obsahuje popis všech 74 tabulek v doménově specifickém jazyce.

Celková délka vygenerovaného mezisouboru činí 2017 řádků. Nárůst velikosti oproti dočasnému souboru probíranému v kapitole 7.1.1 je způsoben především povinným odřádkováním mezi jednotlivými konstrukcemi DSJ, které má uživateli usnadnit orientaci v souboru.

7.1.3 Kontrola validity mezisouboru

Správnost vygenerovaného mezisouboru lze nejlépe otestovat jeho předáním parseru nástroje Seeder. Je však nutné nejdříve ošetřit atributy, jejichž datový typ není tímto nástrojem podporován. V případě Drupalu se jedná pouze o datový typ `bytea`.

Pro takto ošetřený mezisoubor generovaný na základě dumpu Drupalu ani pro mezisoubor žádné z menších testovacích databází nedošlo při zpracování parserem druhým nástrojem k výskytu žádné syntaktické ani semantické chyby.

Korespondence zdrojové databáze a vysledných vygenerovaných dotazů `INSERT` bude řešena v následujících kapitolách.

7.2 Testování správné činnosti nástroje pro generování dat

Narozdíl od prvního nástroje lze správnou činnost nástroje Seeder lépe ověřit pomocí menších testovacích databází zaměřených na všechny konstrukce, které nástroj podporuje.

Nevhodnost databáze Drupalu pro toto testování má několik důvodů. Obsažené typy a konstrukce zdaleka nepokrývají všechny možnosti, které by Seeder měl být schopen naplnit. Dále se jedná o databázi fungujícího systému, která obsahuje mnoho předvyplněných tabulek - kompletní naplnění pomocí nástroje Seeder by v systému mohlo napáchat velké škody. Dají se proto plnit pouze části. K tomu je navíc potřeba, aby se uživatel v plněném fragmentu dostatečně orientoval a dokázal správně upravit mezisoubor - například při plnění cizích klíčů referujících atribut předvyplněné tabulky apod.

7.2.1 Testování validity hodnot generovaných metodou fm_basic()

Pro testování schopnosti nástroje Seeder vygenerovat pro všechny podporované typy data, která databáze přijme, byla navržena testovací databáze 1, dále označovaná jako DB1 (příloha C).

Databáze DB1 se skládá z 22 tabulek, z nichž každá slouží k otestování jednoho podporovaného datového typu. Tento datový typ je nastaven jedinému atributu (`a1`) každé z tabulek.

Výše popsaná databáze byla vytvořena v PostgreSQL databázovém systému a z této byl následně vygenerován její dump. Dump byl dodán na vstup nástroje Harvester s parametrem `--count 100`. Výsledný mezisoubor obsahoval popis všech 22 tabulek s počtem plnění pro každou z nich nastavený na 100.

Po spuštění nástroje Seeder nad tímto mezisouborem došlo k vygenerování 2200 dotazů `INSERT`, které byly uloženy do souboru `all_inserts`. Soubor `all_inserts` byl poté pomocí příkazu

```
psql -f ./all_inserts -d DB1
```

předložen zdrojové databázi.

Tento postup byl proveden několikrát. Při průběhu plnění v žádném z pokusů nedošlo k chybám. Generované hodnoty pro jednotlivé datové typy lze tedy označit za validní.

7.2.2 Testování správného plnění atributů zadaných s integritními omezeními

V průběhu práce na druhém nástroji bylo vytvořeno několik databází, na kterých bylo správné plnění atributů s integritními omezeními testováno. Tyto databáze byly obvykle úzce zaměřené na konkrétní omezení a během ladění v nich docházelo ke změnám.

Pro potřeby této práce byla vytvořena demonstrační databáze DB2 (příloha D). Obsahuje cizí klíče, primární klíče (samostatné atributy i skupiny atributů) a atributy s omezením `UNIQUE`, `NULL` a `DEFAULT`.

Dump databáze DB2 byl opět přiveden na vstup nástroje Harvester, který na jeho základě vygeneroval mezisoubor v DSJ. (Výstup nástroje Harvester pro DB2 je uveden v příloze D.2). Protože tato databáze byla navržena také pro testování správného plnění atributů s omezením `NULL`, bylo nutné v mezisouboru provést změny, které toto testování umožňují[1] (příloha D.3).

Tabulky `primary_key_group` a `unique_key_group` byly navrženy tak, aby vhodně otestovaly schopnost druhého nástroje kontrolovat jedinečnost kombinací více atributů. Pro dvojici atributů obou tabulek byl zvolen datový typ `numeric(1,0)`, který umožňuje vložení jediné cifry, tedy $0 - 9$, čili 10 možných variant. Maximální počet jedinečných kombinací hodnot těchto dvojic je roven součinu $10 * 10$, který je roven číslu 100. Tato hodnota byla zvolena jako počet generovaných dotazů `INSERT` pro všechny tabulky, Seeder proto bude muset vytvořit všechny možné kombinace, aby dokázal požadované dotazy vygenerovat.

Seeder má nastaven interní timeout 100 pro počet pokusů o nalezení unikátní hodnoty či kombinace hodnot, je proto možné, že se generování v těchto hraničních situacích nemusí zadařit napoprvé[2].

Vygenerované dotazy `INSERT` byly použity pro naplnění databáze DB2. Při plnění nedošlo k žádným chybám.

Lze tedy říci, že Seeder dokázal správně ohlídat unikátnost kombinací hodnot pro tabulky `primary_key_group` a `unique_key_group`. Rovněž nedošlo ke kolizi při generování

[1] V SQL se toto omezení považuje za implicitní, v dumpu tedy není vypsáno.

[2] V tomto případě došlo k vypršení timeoutu ve dvou z pěti pokusů.

samostatně unikátních hodnot (`unique` nebo `primary_key`), ani při plnění cizích klíčů již vygenerovanými hodnotami.

Pro omezení `default` s šancí výskytu 20%, které bylo použito pro všechny tři atributy tabulky `default_test`, vyšel poměr hodnoty „DEFAULT“ ku vlastním vygenerovaným hodnotám tak, jak je uvedeno v tabulkách 7.1, 7.2 a 7.3. Průměry získaných hodnot lze nalézt v tabulce 7.4 [3].

Poznámka: Jedná se o poměr oreientační vzhledem k nízkému počtu ověřovaných pokusů.

Atribut	Celkem	Vlastní hodnoty	Hodnota „DEFAULT“
def1	100	88	12
def2	100	90	10
def3	100	87	13

Tabulka 7.1: Plnění za přítomnosti omezení default - pokus č.1

Atribut	Celkem	Vlastní hodnoty	Hodnota „DEFAULT“
def1	100	80	20
def2	100	83	17
def3	100	81	19

Tabulka 7.2: Plnění za přítomnosti omezení default - pokus č.2

Atribut	Celkem	Vlastní hodnoty	Hodnota „DEFAULT“
def1	100	84	16
def2	100	77	23
def3	100	81	19

Tabulka 7.3: Plnění za přítomnosti omezení default - pokus č.3

Na stejném principu jako generování hodnot „DEFAULT“ je založeno také plnění hodnotami „NULL“. Databáze DB2 obsahuje i případ, kdy dochází ke kombinaci těchto dvou omezení (v tomto případě se jedná o zřetězení `null(20)` a `default(20)`). Jak bylo probráno v kapitole 6.6.4, omezení `null` je prioritní a může hodnotou „NULL“ přepsat již vygenerovanou hodnotu „DEFAULT“.

Chování nástroje Seeder pro atribut (`n2`) tabulky `null_test` je zaznamenáno v tabulce 7.5.

7.2.3 Testování zbývajících plnicích metod

Tato část se bude zabývat otestováním použitelnosti zbývajících důležitých plnicích metod, tedy metody `fm_regex()` a `fm_textbank()`. Pro potřeby tohoto testy byla vytvořena jednoduchá databáze DB3 (Příloha E).

[3] Algoritmus pro generování hodnoty DEFAULT je plně závislý na obdržených číslech z modulu random.

	Celkem	Vlastní hodnoty	Hodnota „DEFAULT“
Průměr	100	83	16

Tabulka 7.4: Plnění za přítomnosti omezení default - průměr získaných hodnot

Pokus č.	Celkem	Vlastní hodnoty	„DEFAULT“	„NULL“
1	100	71	9	20
2	100	65	18	17
3	100	69	12	19
4	100	64	10	26
Průměr	100	67	12	20

Tabulka 7.5: Poměr vygenerovaných hodnot při přítomnosti omezení null a default

Databáze DB3 obsahuje pouze jednu tabulku `method_test` se třemi atributy: `name`, `email` a `phone_no`.

Atribut `name` byl použit k otestování funkčnosti metody `fm_textbank()`. Pro potřeby jeho naplnění byla vytvořena textová banka `names.txt` obsahující 20 českých křestních jmen.

Zbývající atributy `email` a `phone_no` umožňují demonstraci rozmanitých možností použití metody `fm_regex()`. Pro naplnění atributu `email` byl použit regulární výraz:

```
([bcdfghjklmnpqrstvwxz][aeiouy]){3}@([bcdfghjklmnpqrstvwxz]
                                                [aeiouy]){2}\.[a-z]{2}
```

na jehož základě je vygenerován řetězec ve formátu e-mailové adresy.

Pro lokální část se generuje řetězec řetězec o šesti znacích na obdobném principu, jako hodnoty pro `VARCHAR` pomocí metody `fm_basic()`, tedy se záměrem připomínat slova. Podobně je regulární výraz navržen i pro doménovou část adresy, pouze se jedná o čtyři znaky následované tečkou a dvojicí libovolných písmenných znaků.

Regulární výraz navržen pro plnění atributu `phone_no` zajišťuje vygenerování telefonního čísla v českém formátu s předvolbou České Republiky +420.

```
\+420( ([0-9]){3}){3}
```

Výsledné hodnoty jsou k nalezení v příloze E.4.

7.2.4 Testování na reálném systému - Drupal

Při testování reálných systémů je potřeba důkladnější zásah do mezisouboru. Jen zřídka je žádoucí naplnění celé databáze, která často obsahuje některé důležité tabulky již předvyplněné (např. pozice ve firmě, pracovní skupiny, kódy oborů, ale i systémová data apod.). V takových případech je nutné z mezisouboru odstranit tabulky, které si nepřejeme plnit, popřípadě nahradit počet plnění klíčovým slovem „FILLED“, odkazují-li na tuto tabulku cizí klíče v tabulkách, které plnit chceme (jak bylo probráno v kapitole 6.2).

Další úpravy mezisouboru se týkají především volby správné plnicí metody nebo integritního omezení, které uživateli umožňují přizpůsobit plněná data vlastní představě. Z toho vyplývá potřeba určité znalosti struktury plněné databáze a významu jejích tabulek.

Jak už bylo řečeno v úvodu kapitoly 7, za účelem otestování nástrojů Harvester a Seeder na reálném systému byl zvolen systém pro správu obsahu Drupal. Drupal pro své potřeby využívá poměrně rozsáhlou, komplexní databázi s množstvím již předvyplněných dat.

Po analýze databáze byl k testovacímu naplnění zvolen fragment týkající se uživatelů systému. Tento fragment se skládá ze tří tabulek, z nichž jedna (`role`) odpovídá popisu předvyplněné tabulky. Tyto tabulky jsou k nalezení v příloze F.

7.2.5 Charakteristika vybraného fragmentu databáze

Jak je z pohledu na zvolené tabulky patrné, obsahují dva prvky, které nástroj Seeder nepodporuje: datový typ `bytea` a integritní omezení `CHECK`.

Omezení `CHECK` parser nástroje Harvester přijímá, při generování mezisouboru jej ale nebere v potaz, protože podpora tohoto omezení v nástroji Seeder zatím není naimplementovaná. Jeho dodržení si uživatel musí ohlídat sám, umožňují-li mu to plnicí metody. V tomto případě stačí použití metody `fm_basic()` nad datovým typem `INT`, která generuje kladná celá čísla.

Datový typ `bytea` parsováním i generováním prochází stejně jako ostatní nepodporované datové typy, v mezisouboru zůstává zapsán minuskami. Je nutné ho ručně přetypovat na podporovaný typ a přidat omezení `null(100)`. Tabulku `user` lze naplnit a následně otestovat i bez naplnění tohoto atributu validními hodnotami.

7.2.6 Vhodná modifikace mezisouboru fragmentu

Po analýze hodnot, které jsou v databázi uloženy pro ručně registrované uživatele, byly na mezisouboru provedeny změny, které generovaná data alespoň částečně přibližují reálným hodnotám. Tyto změny jsou k vidění v příloze F.3.

Modifikace tabulky role:

- Jedná se o předvyplněnou tabulku. Počet plnění nahrazen klíčovým slovem „FILLED“.

Modifikace tabulky users:

- Atribut `uid`: V databázi jsou již 3 uživatelé. Vložení regexu pro generování vyšších hodnot pro `uid`, aby nedošlo ke kolizi.
- Atribut `name`: Snížen limit typu `VARCHAR` na 8.
- Atribut `pass`: Deset znaků jako heslo.
- Atribut `mail`: Stejný regex jako v 7.2.3.
- Atribut `signature_format`: Pomocí regexu přednastaveno na „filtered_html“.
- Atributy `created`, `access` a `login`: Simulovány hodnoty UNIX Time, `default` nastaven na 0%.
- Atribut `status`: Přednastavena hodnota 1 = active user.
- Atribut `timezone`: Přednastavena hodnota „Europe/Berlin“.

- Atributy `theme`, `signature`, `language`, `picture` a `init`: Omezení `default` nastaveno na 100%.
- Atribut `data`: Typ změněn na `INT`, dodáno omezení `null(100)`.

Modifikace tabulky users_roles:

- U obou atributů změněno plnění na cizí klíče referující do tabulek `role` a `users`. `Default` nastaven na 0%.

Tabulka `users_roles` je klasickým produktem vazby $m : n$ mezi tabulkami `role` a `users`. Zajištění jejího plnění unikátními hodnotami těchto tabulek je zřejmě řešeno mimo databázi, proto bylo nutné do mezisouboru tuto závislost dodat (metoda `fm_reference()`).

Pokud by si uživatel přál databázi naplnit pouze „registrovanými“ uživateli, může pomocí `fm_regex()` přednastavit hodnotu, kterou je plněn atribut `rid` tabulky `users_roles`. Zde je ponechán cizí klíč do předvyplněné tabulky kvůli demonstraci této funkce nástroje Seeder.

7.2.7 Výsledky plnění

Vygenerované hodnoty, jejichž vzorek je k dispozici v příloze F.4, byly vloženy do databáze. Až na drobný problém popsaný v následujícím odstavci proběhlo plnění bez problémů.

Zmíněným problémem je nevhodnost zvoleného postupu pro plnění atributu `rid` tabulky `users_roles`. Vkládaný `SELECT` pro získání hodnoty z předvyplněné tabulky ve 2/10 případů vrátil hodnotu `NULL` a tyto konkrétní dotazy `INSERT` bylo potřeba provést znovu. Důvod této neočekávané situace může ležet ve strukturách databáze, které pro tuto práci mají status tzv. „Black box“, nebo se může jednat o nezmapovanou chybu v konstrukci zanořeného dotazu `SELECT`, kterou nástroj Seeder používá.

Po úspěšném naplnění databáze Drupalu všemi deseti vygenerovanými uživateli a jejich rolemi, byla jejich přítomnost ověřena v grafickém uživatelském rozhraní systému. Výsledky je možné nalézt v příloze F.5

7.3 Testování možnosti automatizace nástrojů

Vhodnost použití nástrojů při automatizaci (se zásahem do mezisouboru i bez) byla zkoušena v průběhu předchozích testů.

Kapitola 8

Závěr

V rámci této bakalářské práce došlo k navržení a implementaci dvojice nástrojů, jejichž společným použitím má být uživateli usnadněno generování testovacích dat pro databázové servery PostgreSQL. První nástroj slouží k analýze dodaného databázového dumpu a následnému generování mezisouboru v doménově specifickém jazyce. Druhý nástroj tento soubor přijímá na vstupu a na jeho základě generuje hodnoty k plnění cílové databáze.

Generování dat se řídí popisem obsaženém v mezisouboru, který je výstupem prvního nástroje, nebo je vytvořen samotným uživatelem na základě dodaných pravidel. Jedná se o textový soubor psaný ve vhodně navrženém doménově specifickém jazyce. Úpravou mezisouboru může uživatel blíže specifikovat vlastnosti generovaných dat, např. zvolit nejvhodnější plnicí metodu či nastavit výskyt hodnot „NULL“ nebo „DEFAULT“.

Nástroje byly otestovány na databázích pokrývajících podporované datové typy a integritní omezení. Dále byla jejich činnost vyzkoušena při naplnění fragmentu databáze reálného systému, v tomto případě se jednalo o systém pro správu obsahu Drupal.

Prostor pro další rozšíření je u těchto nástrojů značný. Analýza databázového dumpu, kterou provádí první nástroj, může být komplexnější. Může brát v potaz více konstrukcí zdrojového SQL, včetně již uložených dat, a na jejich základě lépe vybírat předgenerované plnicí metody. Je také možné celý nástroj modifikovat tak, aby potřebná data získával přímo z vlastního připojení k databázi. Vítaným rozšířením by nepochybně byla i podpora dalších databázových systémů (např. MySQL, Oracle).

Ještě více možností se nabízí u druhého nástroje, potažmo doménově specifického jazyka. Mezi tyto možnosti patří např. plnění dosud nepodporovaných datových typů, podpora integritního omezení `CHECK`, větší škála zabudovaných plnicích metod nebo umožnění tvorby vlastních plnicích metod. Lze také přidat nové konstrukce doménově specifického jazyka, které by uživateli umožnily přesněji specifikovat vzájemnou závislost mezi atributy. Podobně jako první nástroj lze i tento přizpůsobit k přímé komunikaci s plněnou databází. Nástroj v současné podobě zajišťuje dodržení integritních omezení pouze v rámci hodnot, které generuje sám. Může tedy dojít ke kolizi s hodnotami, které již v databázi jsou, ale nástroj o jejich existenci neví. Získávání informací přímo z plněné databáze by podobným kolizím zamezilo.

Oba implementované nástroje jsou zcela ovládány přes příkazový řádek a specifikace plnění závisí na editaci textového souboru. Tyto vlastnosti činí produkty této bakalářské práce vhodné k použití při automatizaci.

Literatura

[1] Adam Tauber: exrex – exrex 0.5.2 documentation. `http://exrex.readthedocs.org/en/latest/`, 2012 [cit. 2014-04-28].

[2] Ben Keen: generatedata.com. `http://www.generatedata.com/`, 2005-2014 [cit. 2014-05-16].

[3] Daryl Harms, K. M.: *ZaĂ¨ĂnĂAme programovat v jazyce Python*. Computer Press, a.s., druhĂŠ opravnĂŠ vydání, 2008, ISBN 978-80-251-2161-0.

[4] Datanamic Data Generator MultiDB. `http://www.datanamic.com/datagenerator/features.html`, 1999-2014 [cit. 2014-05-16], Datanamic Solutions BV.

[5] David M. Beazley: PLY (Python Lex-Yacc). `http://www.dabeaz.com/ply/ply.html`, 2001-2011 [cit. 2014-05-02].

[6] Fowler, M.; Parson, R.: *Domain-Specific Languages*. Addison-Wesley Professional, prvnĂ vydání, 2011, ISBN 0-321-71294-3.

[7] Kahan, W.: IEEE Standard 754 for Binary Floating-Point Arithmetic. Standard, Elect. Eng. Computer Science, University of California, Berkeley CA 94720-1776, oct 1997.

[8] Generating Test Data for Databases by Using Data Generators. `http://msdn.microsoft.com/en-us/library/dd193262`, 2014 [cit. 2014-05-16], Microsoft.

[9] Pete Jinks: Notations for context-free grammars. `http://www.cs.man.ac.uk/ pjj/bnf/bnf.html`, 2004-03-12 [cit. 2014-03-22].

[10] PostgreSQL : Documentation. `http://www.postgresql.org/docs/`, 1996-2014 [cit. 2014-05-17], The PostgreSQL Global Development Group.

[11] Python v2.7.6 documentation. `http://exrex.readthedocs.org/en/latest/`, 1990-2014 [cit. 2014-04-08], Python Software Foundation.

[12] SQL Data Generator - Data generator for MS SQL Server databases. `http://www.red-gate.com/products/sql-development/sql-data-generator/`, 1999-2014 [cit. 2014-05-16], Red Gate Software, Inc.

Příloha A

Návod k instalaci a otestování nástrojů

Součástí práce je disk DVD s obrazem virtuálního stoje s operačním systémem Fedora 19. Tento stroj byl připraven tak, aby umožnil co nejsnadnější otestování nástrojů vyvinutých v rámci této práce. (Disk by měl být k nalezení z vnitřní strany zadních desek.)

Stroj obsahuje oba nástroje, nainstalovaný server PostgreSQL a všechny prerekvizity pro spuštění systému Drupal verze 7.2.8.

Tato příloha se dále zabývá postupem při instalaci a testování.

A.1 Postup při instalaci a spuštění systému

1. Rozbalte obraz virtuálního stroje, který je k dispozici na přiloženém disku DVD v archivu **BP_testing.rar**.
2. Pro jeho spuštění se předpokládá použití virtualizačního nástroje **Virtual Box**. Spusťte jej a v nabídce v levém horním rohu zvolte možnost *New*.
3. Zvolte jméno a systém virtuálního stroje (System: Linux, Version: Fedora19 (64bit)).
4. Velikost RAM nastavte na 1024MB.
5. V následujícím okně zvolte variantu „Use an existing virtual hard drive file“ a najděte cestu k rozbalenému souboru **BP_testing.vdi**.
6. Potvrzením dojde k vytvoření virtuálního stroje k testování. Spusťte jej.
7. Pro uživatele **tester** zadejte heslo: *tester* a vstupte do systému.

A.2 Postup při testování nástrojů na uměle navržených databázích

1. V terminálu se přesuňte do složky, ze které bude testování probíhat:

   ```
   cd /home/tester/sql-data-farmer/
   ```

2. Pro testování nástrojů, které je popsáno v kapitole 7, jsou v systému připravené tři databáze pro tři testovací schémata (přílohy C.1, D.1 a E.1). Tato schémata jsou k dispozici v adresáři `/home/tester/sql-data-farmer/testdbs/`. Nejprve do první

databáze `test` nahrajeme schéma `all_db` pro testování generování kompatibilních dat pro plnění:

```
psql -f ./testdbs/all_db -d test
```

3. Následně vygenerujeme dump této databáze, který uložíme do složky `testdbs/`:

```
pg_dump test > ./testdbs/all_dump
```

4. Získaný dump předložíme na vstupu nástroji Harvester s parametrem `-c` nastavujícím počet generovaných insertů každé tabulky na 100. Přepínač `-f` způsobí uložení mezisouboru do domovského adresáře uživatele pod názvem `dsl.txt`, tedy z pracovního adresáře na cestě `../dsl.txt`.

```
./Harvester/harvester.py ./testdbs/all_dump -f -c 100
```

5. Mezisoubor následně pošleme na vstup nástroje Seeder a výsledný skript s dotazy `INSERT` uložíme opět do složky `testdbs/` pod názvem `all_inserts`. (Spuštění nástroje bez přepínače znamená, že nástroj počítá s přítomností souboru `dsl.txt` v domovském adresáři uživatele.)

```
./Seeder/seeder.py > ./testdbs/all_inserts
```

6. Konečně použijeme získaný skript k naplnění databáze **test**:

```
psql -f ./testdbs/all_inserts -d test
```

7. Nedošlo-li při některém z 2200 vkládání k žádné chybě způsobené nekompatibilitou vkládaných dat, testování na databázi `test1` proběhlo úspěšně.
8. Vložené hodnoty si lze prohlédnout použitím patřičného dotazu za parametrem `-c` příkazu `psql` např. takto:

```
psql -c 'select * from t_varchar;' -d test
```

9. Uvedení databáze do původního, nenaplněného stavu pro opakované otestování (postup se následně opakuje od bodu 3):

```
psql -c 'drop schema public cascade;' -d test
psql -c 'create schema public;' -d test
psql -f ./testdbs/all_db -d test
```

Stejný postup je použit pro naplnění zbývajících dvou testovacích databází s následujícími sadami příkazů

Databáze pro otestování integritních omezení - test2

1. `cd /home/tester/sql-data-farmer/`
2. `psql -f ./testdbs/constraint_test_db -d test2`
3. `pg_dump test2 > ./testdbs/constraint_test_dump`
4. `./Harvester/harvester.py ./testdbs/constraint_test_dump -f -c 50`
5. `./Seeder/seeder.py > ./testdbs/constraint_test_inserts`
6. `psql -f ./testdbs/constraint_test_inserts -d test2`
7. Zobrazení naplněných tabulek. Např:

```
psql -c 'select * from unique_group;' -d test2
```

8. Původní stav (schéma vloženo, tabulky prázdné):

```
psql -c 'drop schema public cascade;' -d test2
psql -c 'create schema public;' -d test2
psql -f ./testdbs/constraint_test_db -d test2
```

Databáze pro otestování plnicích metod fm_regex() a fm_textbank() - test3

1. `cd /home/tester/sql-data-farmer/`
2. `psql -f ./testdbs/metody_test_db -d test3`
3. `pg_dump test3 > ./testdbs/metody_test_dump`
4. `./Harvester/harvester.py ./testdbs/metody_test_dump -f -c 50`
5. `./Seeder/seeder.py > ./testdbs/metody_test_inserts`
6. `psql -f ./testdbs/metody_test_inserts -d test3`
7. Zobrazení naplněných tabulek. Např:

```
psql -c 'select * from method_test;' -d test3
```

8. Původní stav (schéma vloženo, tabulky prázdné):

```
psql -c 'drop schema public cascade;' -d test3
psql -c 'create schema public;' -d test3
psql -f ./testdbs/metody_test_db -d test3
```

Odstranění mezisouborů a souborů s plnicími skripty

```
cd /home/tester/sql-data-farmer/
rm ./testdbs/*_dump ./testdbs/*_inserts
```

A.3 Postup při testování nástrojů na na fragmentu databáze systému Drupal

1. Pracovní složka zůstává stejná jako výše:

   ```
   cd /home/tester/sql-data-farmer/
   ```

2. Systém Drupal je již ve virtuálním stroji nainstalován a připraven k testování. Může se tedy rovnou přejít na vygenerování dumpu jeho databáze. Dump bude uložen do složky `/home/tester/sql-data-farmer/drupal/`.

   ```
   pg_dump drupal > ./drupal/drupal_dump
   ```

3. Schopnost Harvestera otestovat dump rozsáhlejší reálné databáze otestujeme podobně jako v předchozích případech:

   ```
   ./Harvester/harvester.py ./drupal/drupal_dump -f
   ```

4. Jak bylo rozebráno v kapitole 7.2.4, vygenerovaný mezisoubor v této chvíli uložený v souboru `../dsl.txt`, nelze z několika důvodů použít. Spuštění nástroje Seeder následujícím způsobem skončí chybou parseru z důvodu přítomnosti nepodporovaného datového typu (`bytea`) v mezisouboru:

   ```
   ./Seeder/seeder.py > ./drupal/drupal_inserts
   ```

   Z důvodů popsaných v kapitole 7.2.4 je testování v tomto případě prováděno pouze na fragmentu cílové databáze. Již pozměněný mezisoubor se nachází ve složce `drupal/` pod názvem `fragment_dsl.txt`. Mezisoubor se mírně liší od mezisouboru uvedeného v příloze F.3 z důvodů problému popsaného v kapitole 7.2.7. Atribut `rid` tabulky `users_roles` je zde plněn přednastavenou hodnotou značící registrovaného uživatele (číslo 1), přítomnost tabulky `role` nastavené jako předvyplněná tedy není nutný.

   Vygenerování plnicího skriptu se tedy provede následujícím příkazem:

   ```
   ./Seeder/seeder.py > ./drupal/drupal_inserts -s\
                                          ./drupal/fragment_dsl.txt
   ```

5. Výsledný skript poté použijeme:

   ```
   psql -f ./drupal/drupal_inserts -d drupal
   ```

6. Vygenerované hodnoty si lze prohlédnout přímo v databázi podobně jako při předchozím testování. Např.:

   ```
   psql -c 'select * from users;' -d drupal
   ```

7. Názornější je ukázka v samotném GUI Drupalu. Přes prohlížeč (např. Firefox na ploše) se připojte na adresu:

   ```
   localhost/drupal/
   ```

8. Uživatel, který je pro manipulaci s GUI nastaven, má jméno **admin**, heslo *admin*.
9. V levé horní části drupalu se nachází menu. V tomto případě je důležitá patá textová položka zleva, tedy **People**. Dojde k zobrazení uživatelů existujících v systému, v tomto případě se vyjma uživatele **admin** jedná o uživatele vzniklé použitím vygenerovaného skriptu.

   Poznámka: V horní části okna může dojít k zobrazení varování Drupalu týkajícího se výskytu nedefinovaného offsetu. Jedná se o důsledek příliš mělké znalosti plněné databáze a z něj vyplývající neschopnosti zaručit vlastním zásahem 100% validnost dat. V reálném nasazení se předpokládá, že uživatel se v testované databázi a systému vyzná dostatečně.
10. Návrat k původnímu stavu databáze (tj. pouze uživatel **admin**):

    ```
    psql -c 'delete from users_roles where uid > 1;' -d drupal
    psql -c 'delete from users where uid > 1;' -d drupal
    ```

Odstranění mezisouboru a dumpu

```
cd /home/tester/sql-data-farmer/
rm ./drupal/*_dump ./testdbs/*_inserts
```

Příloha B

Gramatika navrženého doménově specifického jazyka

```
<dsl> ::= <tableBlock> <moreBlocks>

<moreBlocks> ::= <moreBlocks> <tableBlock> | <empty>
<tableBlock> ::= <tableHeader> <attributeBlock> <moreAttributes>
<tableHeader> ::=  TABLE COLON IDENTIFIER LPAREN NUMBER RPAREN <endline>
                |  TABLE COLON IDENTIFIER LPAREN FILLED RPAREN <endline>

<moreAttributes> ::= <moreAttributes> <attributeBlock> | <empty>
<attributeBlock> ::= <attributeName> <dataType> <fillMethod>
                   | <attributeName> <dataType> <fillMethod> <constraintPart>
<attributeName> ::= DOUBLE_COLON IDENTIFIER <endline>


<dataType> ::= TYPE <dtypes> <moreDimensions> <endline>
<dtypes> ::= TYPE_NOPARAM
           | TYPE_1PARAM LPAREN NUMBER RPAREN
           | TYPE_2PARAM LPAREN NUMBER COMMA NUMBER RPAREN
           | TYPE_2PARAM_TIME LPAREN NUMBER COMMA TIMEZONE_PARAM RPAREN

<moreDimensions> ::= <moreDimensions> <oneDimension> | <empty>
<oneDimension> ::= LBRACKET NUMBER RBRACKET

<fillMethod> ::= FILL FILL_METHOD_NOPARAM LPAREN RPAREN <endline>
               | FILL FILL_METHOD_1PARAM LPAREN <parameter> RPAREN <endline>

<constraintPart> ::= CONSTRAINT <constr> <moreConstr> <endline>
<moreConstr> ::= <constr> <moreConstr> | <empty>
<constr> ::= CONSTR_NOPARAM
           | CONSTR_1PARAM LPAREN NUMBER RPAREN

<parameters> ::= <parameters> <1parameter> | <empty>
<1parameter> ::= PATH | REGEX | NUMBER | IDENTIFIER
```

```
                | IDENTIFIER COLON IDENTIFIER

<endline> ::= EOL <extraEndline>
<extraEndline> ::= EOL <extraEndline> | <empty>
<empty> ::=

TABLE ::= "TABLE"
TYPE ::= "TYPE"
FILL ::= "FILL"
CONSTRAINT ::= "CONSTRAINT"
FILLED ::= "FILLED"

CONSTR_NOPARAM ::= "foreign_key" | "not_null"
CONSTR_1PARAM ::= "unique" | "primary_key" | "null" | "default"

FILL_METHOD_NOPARAM ::= "fm_basic"
FILL_METHOD_1PARAM ::= "fm_regex" | "fm_textbank" | "fm_reference"

TYPE_NOPARAM ::= "BIGINT" | "BIGSERIAL" | "BOOL" | "BOX" |
                 "CIDR" | "CIRCLE" | "DATE" | "DOUBLE" |
                 "INET" | "INT" | "LSEG" | "MACADDR" | "PATH" |
                 "POINT" | "POLYGON" | "REAL" | "SERIAL" |
                 "SMALLINT" | "TEXT"
TYPE_1PARAM ::= "BIT" | "CHAR" | "VARBIT" | "VARCHAR"
TYPE_2PARAM ::= "NUMERIC"
TYPE_2PARAM_TIME ::= "TIME" | "TIMESTAMP"

DOUBLE_COLON ::= "::"
COLON  ::= ":"
LPAREN  ::= "("
RPAREN  ::= ")"
COMMA ::= ","
LBRACKET = "["
RBRACKET = "]"
QUOTE = "\"
TIMEZONE_PARAM ::= "+TMZ" | "-TMZ"
REGEX ::= '[\ 0-9A-Za-z#$%=@!{},‘~&*()<>?.:;_|^/+[]"-]+'
IDENTIFIER ::= [a-zA-Z][a-zA-Z_0-9]*
NUMBER ::= (-)?[0-9]+
EOL ::= "\n"
PATH ::= [a-zA-Z_0-9/-.]+
```

Příloha C

Testovací databáze 1

Databáze (označením DB1) použitá v kapitole 7 pro otestování validity dat generovaných pro podporované datové typy pomocí metody `fm_basic()`.

C.1 Databáze DB1

```
CREATE TABLE t_bigint (
    a1 bigint
);
CREATE TABLE t_bigserial (
    a1 bigserial NOT NULL
);
CREATE TABLE t_bit (
    a1 bit(4)
);
CREATE TABLE t_bit_var (
    a1 bit varying(8)
);
CREATE TABLE t_bool (
    a1 boolean
);
CREATE TABLE t_box (
    a1 box
);
CREATE TABLE t_char (
    a1 character(5)
);
CREATE TABLE t_cidr (
    a1 cidr
);
CREATE TABLE t_circle (
    a1 circle
);
CREATE TABLE t_double_prec (
    a1 double precision
```

```
);
CREATE TABLE t_inet (
    a1 inet
);
CREATE TABLE t_integer (
    a1 integer
);
CREATE TABLE t_lseg (
    a1 lseg
);
CREATE TABLE t_numeric (
    a1 numeric(5,2)
);
CREATE TABLE t_path (
    a1 path
);
CREATE TABLE t_point (
    a1 point
);
CREATE TABLE t_polygon (
    a1 polygon
);
CREATE TABLE t_real (
    a1 real
);
CREATE TABLE t_serial (
    a1 serial NOT NULL
);
CREATE TABLE t_smallint (
    a1 smallint
);
CREATE TABLE t_text (
    a1 text
);
CREATE TABLE t_varchar (
    a1 character varying(5)
);
```

Příloha D

Testovací databáze 2

Databáze (označením DB2) použitá v kapitole 7 pro testování správného plnění atributů s integritními omezeními `PRIMARY KEY`, `UNIQUE`, `FOREIGN KEY`, `DEFAULT` a `NULL`.

Tato příloha zahrnuje také obsah mezisouboru, který byl výstupem nástroje Harvester pro tuto databázi (sekce D.2) a inserty s daty vygenerované nástrojem Seeder D.4.

D.1 Databáze DB2

```
CREATE TABLE primary_key_single(
pk char(2) PRIMARY KEY
);

CREATE TABLE default_test(
defpk integer PRIMARY KEY,
def1 integer DEFAULT(1),
def2 varchar DEFAULT'testing',
def3 text    DEFAULT''
);

CREATE TABLE primary_key_group(
pk_group1 numeric(1,0),
pk_group2 numeric(1,0)
);

ALTER TABLE ONLY primary_key_group
    ADD CONSTRAINT test_pk PRIMARY KEY (pk_group1, pk_group2);

CREATE TABLE unique_group(
uq_group1 numeric(1,0),
uq_group2 numeric(1,0)
);

ALTER TABLE ONLY unique_group
    ADD CONSTRAINT test_uq UNIQUE (uq_group1, uq_group2);
```

```
CREATE TABLE foreign_key_test(
fk1 char(2) references primary_key_single(pk),
fk2 integer references default_test(defpk)
);

CREATE TABLE null_test(
n1 integer NULL,
n2 double precision NULL default(0.0)
);
```

D.2 Mezisoubor vygenerovaný pro DB2

Poznámka: Constraint `NULL` se v SQL považuje za implicitní, dump databáze ho tedy neobsahuje a nemůže být tudíž přítomen ani ve vygenerovaném mezisouboru. Chceme-li hodnotami „NULL“ plnit, je nutné mezisoubor modifikovat.

Modifikace zdokumentována v části D.3.

```
TABLE:default_test(100)
        ::defpk
                TYPE INT
                FILL fm_basic()
                CONSTRAINT primary_key(0) not_null
        ::def1
                TYPE INT
                FILL fm_basic()
                CONSTRAINT default(20)
        ::def2
                TYPE VARCHAR(8)
                FILL fm_basic()
                CONSTRAINT default(20)
        ::def3
                TYPE TEXT
                FILL fm_basic()
                CONSTRAINT default(20)

TABLE:foreign_key_test(100)
        ::fk1
                TYPE CHAR(2)
                FILL fm_reference(primary_key_single:pk)
                CONSTRAINT foreign_key
        ::fk2
                TYPE INT
                FILL fm_reference(default_test:defpk)
                CONSTRAINT foreign_key

TABLE:null_test(100)
        ::n1
                TYPE INT
```

```
                FILL fm_basic()
        ::n2
                TYPE DOUBLE
                FILL fm_basic()
                CONSTRAINT default(20)

TABLE:primary_key_group(100)
        ::pk_group1
                TYPE NUMERIC(1,0)
                FILL fm_basic()
                CONSTRAINT primary_key(1) not_null
        ::pk_group2
                TYPE NUMERIC(1,0)
                FILL fm_basic()
                CONSTRAINT primary_key(1) not_null

TABLE:primary_key_single(100)
        ::pk
                TYPE CHAR(2)
                FILL fm_basic()
                CONSTRAINT primary_key(0) not_null

TABLE:unique_group(100)
        ::uq_group1
                TYPE NUMERIC(1,0)
                FILL fm_basic()
                CONSTRAINT unique(2)
        ::uq_group2
                TYPE NUMERIC(1,0)
                FILL fm_basic()
                CONSTRAINT unique(2)
```

D.3 Modifikace mezisouboru DB2 pro účely testování

Původní konstrukce:

```
TABLE:null_test(100)
        ::n1
                TYPE INT
                FILL fm_basic()
        ::n2
                TYPE DOUBLE
                FILL fm_basic()
                CONSTRAINT default(20)
```

Modifikovaná konstrukce:

```
TABLE:null_test(100)
        ::n1
                TYPE INT
                FILL fm_basic()
                CONSTRAINT null(50)
        ::n2
                TYPE DOUBLE
                FILL fm_basic()
                CONSTRAINT default(20) null(20)
```

D.4 Vzorek vygenerovaných dat pro plnění DB2

Poskytnutí všech insertů získaných po průchodu meziouboru nástrojem Seeder je z prostorových důvodů nemožné. Bude zde proto demonstrován pouze malý vzorek obdržených dat.

Pro lepší demonstraci závislosti cizích klíčů na referovaných atributech byla za účelem tohoto vzorku vygenerována nová sada insertů. Její předlohou byl mezisoubor D.2 s počtem generovaných insertů sníženým na 5.

(Věty vygenerované pro atribut datového typu `TEXT` byly zkráceny z důvodu správného vysázení.)

```
INSERT INTO default_test
VALUES (95, 584479340, DEFAULT, DEFAULT);

INSERT INTO default_test
VALUES (255272979, 357641527, 'Nojuweda', 'American occupying forces leave.');

INSERT INTO default_test
VALUES (381343900, 2819592, 'Copunat', DEFAULT);

INSERT INTO default_test
VALUES (873490043, 502550, 'Feqemovy', 'Authority votes to decertify.');

INSERT INTO default_test
VALUES (49490236, 899027, 'Xubu', DEFAULT);

INSERT INTO null_test
VALUES (8283, NULL);

INSERT INTO null_test
VALUES (1641425627, NULL);

INSERT INTO null_test
VALUES (NULL, 3.029e-306);

INSERT INTO null_test
VALUES (949428967, 9.803e-83);
```

```
INSERT INTO null_test
VALUES (NULL, 6.885e-15);

INSERT INTO primary_key_group
VALUES (8, 9);

INSERT INTO primary_key_group
VALUES (1, 4);

INSERT INTO primary_key_group
VALUES (3, 8);

INSERT INTO primary_key_group
VALUES (5, 2);

INSERT INTO primary_key_group
VALUES (2, 2);

INSERT INTO primary_key_single
VALUES ('XL');

INSERT INTO primary_key_single
VALUES ('Dc');

INSERT INTO primary_key_single
VALUES ('gp');

INSERT INTO primary_key_single
VALUES ('q3');

INSERT INTO primary_key_single
VALUES ('k5');

INSERT INTO unique_group
VALUES (5, 5);

INSERT INTO unique_group
VALUES (1, 6);

INSERT INTO unique_group
VALUES (0, 3);

INSERT INTO unique_group
VALUES (8, 4);

INSERT INTO unique_group
VALUES (5, 4);
```

```
INSERT INTO foreign_key_test
VALUES ('gp', 381343900);

INSERT INTO foreign_key_test
VALUES ('XL', 873490043);

INSERT INTO foreign_key_test
VALUES ('Dc', 381343900);

INSERT INTO foreign_key_test
VALUES ('q3', 873490043);

INSERT INTO foreign_key_test
VALUES ('Dc', 49490236);
```

Příloha E

Testovací databáze 3

Jednoduchá databáze obsahující pouze jednu tabulku `method_test`. Slouží k demonstraci použitelnosti metod `fm_regex()` a `fm_textbank()`.

E.1 Databáze DB3

```
CREATE TABLE method_test(
name varchar(20),
email text,
phone_no char(16)
);
```

E.2 Mezisoubor vygenerovaný pro DB3

```
TABLE:method_test(10)
        ::name
                TYPE VARCHAR(20)
                FILL fm_basic()

        ::email
                TYPE TEXT
                FILL fm_basic()

        ::phone_no
                TYPE CHAR(16)
                FILL fm_basic()
```

E.3 Modifikace mezisouboru DB3 pro účely testování

Poznámka: Textová banka `names.txt` byla vytvořena v adresáři, ze kterého byly nástroje Harvester a Seeder spouštěny.

Celý regulární výraz použitý pro plnění atributu `email` má následující tvar:

```
([bcdfghjklmnpqrstvwxz][aeiouy]){3}@([bcdfghjklmnpqrstvwxz]
                                                [aeiouy]){2}\.[a-z]{2}
```

Konstrukce mezisouboru pro DB3:

```
TABLE:method_test(10)
        ::name
                TYPE VARCHAR(20)
                FILL fm_textbank(./names.txt)

        ::email
                TYPE TEXT
                FILL fm_regex('([bcdfghjklmnpqrstvwxz] ... \.[a-z]{2}')

        ::phone_no
                TYPE CHAR(16)
                FILL fm_regex('\+420( ([0-9]){3}){3}')
```

E.4 Vzorek vygenerovaných dat pro plnění DB3

```
INSERT INTO method_test
VALUES ('Pavel', 'tilixi@qizo.ei', '+420 105 567 416');

INSERT INTO method_test
VALUES ('Jana', 'konopo@tuma.nq', '+420 036 002 860');

INSERT INTO method_test
VALUES ('Jana', 'movyha@kimi.cb', '+420 147 430 011');

INSERT INTO method_test
VALUES ('Gabriela', 'kalimo@weja.ob', '+420 786 726 172');

INSERT INTO method_test
VALUES ('Andrea', 'wuzihu@quca.um', '+420 432 946 636');

INSERT INTO method_test
VALUES ('Alena', 'xyvyvi@pore.ki', '+420 183 304 446');

INSERT INTO method_test
VALUES ('Tomáš', 'tejave@riza.lo', '+420 668 447 932');

INSERT INTO method_test
VALUES ('Petra', 'kabeti@pini.px', '+420 699 622 557');

INSERT INTO method_test
VALUES ('Hana', 'qodimi@jahu.li', '+420 806 559 848');

INSERT INTO method_test
VALUES ('Jitka', 'mexeja@vuqo.hd', '+420 077 387 263');
```

Příloha F

Databáze Drupalu

Systém Drupal byl zvolen pro otestování nástrojů na reálném systému. Testování bude probíhat na vybraném fragmentu.

F.1 Testovaný fragment databáze Drupalu

Zvolený fragment databáze Drupalu, jak byl získán z databázového dumpu. (Pozn.: Z prostorových důvodů byly z některých atributů odstraněny notace přetypování `::character varying`, tyto však nejsou pro demonstraci relevantní.)

```
CREATE TABLE role (
    rid integer NOT NULL,
    name character varying(64) DEFAULT '' NOT NULL,
    weight integer DEFAULT 0 NOT NULL,
    CONSTRAINT role_rid_check CHECK ((rid >= 0))
);

CREATE TABLE users (
    uid bigint DEFAULT 0 NOT NULL,
    name character varying(60) DEFAULT '' NOT NULL,
    pass character varying(128) DEFAULT '' NOT NULL,
    mail character varying(254) DEFAULT '',
    theme character varying(255) DEFAULT '' NOT NULL,
    signature character varying(255) DEFAULT '' NOT NULL,
    signature_format character varying(255),
    created integer DEFAULT 0 NOT NULL,
    access integer DEFAULT 0 NOT NULL,
    login integer DEFAULT 0 NOT NULL,
    status smallint DEFAULT 0 NOT NULL,
    timezone character varying(32),
    language character varying(12) DEFAULT '' NOT NULL,
    picture integer DEFAULT 0 NOT NULL,
    init character varying(254) DEFAULT '',
    data bytea,
    CONSTRAINT users_uid_check CHECK ((uid >= 0))
);
```

```
CREATE TABLE users_roles (
    uid bigint DEFAULT 0 NOT NULL,
    rid bigint DEFAULT 0 NOT NULL,
    CONSTRAINT users_roles_rid_check CHECK ((rid >= 0)),
    CONSTRAINT users_roles_uid_check CHECK ((uid >= 0))
);

ALTER TABLE ONLY users_roles
    ADD CONSTRAINT users_roles_pkey PRIMARY KEY (uid, rid);
```

F.2 Mezisoubor vygenerovaný pro zvolený fragment

```
TABLE:role(10)
        ::rid
                TYPE SERIAL
                FILL fm_basic()
                CONSTRAINT primary_key(0) not_null
        ::name
                TYPE VARCHAR(64)
                FILL fm_basic()
                CONSTRAINT unique(0) not_null default(20)
        ::weight
                TYPE INT
                FILL fm_basic()
                CONSTRAINT not_null default(20)

TABLE:users(10)
        ::uid
                TYPE BIGINT
                FILL fm_basic()
                CONSTRAINT primary_key(0) not_null default(20)
        ::name
                TYPE VARCHAR(60)
                FILL fm_basic()
                CONSTRAINT unique(0) not_null default(20)
        ::pass
                TYPE VARCHAR(128)
                FILL fm_basic()
                CONSTRAINT not_null default(20)
        ::mail
                TYPE VARCHAR(254)
                FILL fm_basic()
                CONSTRAINT default(20)
        ::theme
                TYPE VARCHAR(255)
                FILL fm_basic()
```

```
                CONSTRAINT not_null default(20)
        ::signature
                TYPE VARCHAR(255)
                FILL fm_basic()
                CONSTRAINT not_null default(20)
        ::signature_format
                TYPE VARCHAR(255)
                FILL fm_basic()
        ::created
                TYPE INT
                FILL fm_basic()
                CONSTRAINT not_null default(20)
        ::access
                TYPE INT
                FILL fm_basic()
                CONSTRAINT not_null default(20)
        ::login
                TYPE INT
                FILL fm_basic()
                CONSTRAINT not_null default(20)
        ::status
                TYPE SMALLINT
                FILL fm_basic()
                CONSTRAINT not_null default(20)
        ::timezone
                TYPE VARCHAR(32)
                FILL fm_basic()
        ::language
                TYPE VARCHAR(12)
                FILL fm_basic()
                CONSTRAINT not_null default(20)
        ::picture
                TYPE INT
                FILL fm_basic()
                CONSTRAINT not_null default(20)
        ::init
                TYPE VARCHAR(254)
                FILL fm_basic()
                CONSTRAINT default(20)
        ::data
                TYPE bytea
                FILL fm_basic()

TABLE:users_roles(10)
        ::uid
                TYPE BIGINT
                FILL fm_basic()
                CONSTRAINT primary_key(26) not_null default(20)
```

```
        ::rid
                TYPE BIGINT
                FILL fm_basic()
                CONSTRAINT primary_key(26) not_null default(20)
```

F.3 Modifikace mezisouboru fragmentu pro účely testování

```
TABLE:role(FILLED)
        ::rid
                TYPE SERIAL
                FILL fm_basic()
                CONSTRAINT primary_key(0) not_null
        ::name
                TYPE VARCHAR(64)
                FILL fm_basic()
                CONSTRAINT unique(0) not_null default(20)
        ::weight
                TYPE INT
                FILL fm_basic()
                CONSTRAINT not_null default(20)

TABLE:users(10)
        ::uid
                TYPE BIGINT
                FILL fm_regex('12[0-9]{2}')
                CONSTRAINT primary_key(0) not_null default(0)
        ::name
                TYPE VARCHAR(8)
                FILL fm_basic()
                CONSTRAINT unique(0) not_null default(0)
        ::pass
                TYPE VARCHAR(128)
                FILL fm_regex('[A-Za-z0-9_\!\%\$\#\@]{10}')
                CONSTRAINT not_null default(20)
        ::mail
                TYPE VARCHAR(254)
                FILL fm_regex('([bcdfghjklmnpqrstvwxz] ... \.[a-z]{2}')
                CONSTRAINT default(0)
        ::theme
                TYPE VARCHAR(255)
                FILL fm_basic()
                CONSTRAINT not_null default(100)
        ::signature
                TYPE VARCHAR(255)
                FILL fm_basic()
                CONSTRAINT not_null default(100)
        ::signature_format
                TYPE VARCHAR(255)
```

```
                FILL fm_regex('filtered_html')
        ::created
                TYPE INT
                FILL fm_regex('131[0-9]{7}')
                CONSTRAINT not_null default(0)
        ::access
                TYPE INT
                FILL fm_regex('132[0-9]{7}')
                CONSTRAINT not_null default(0)
        ::login
                TYPE INT
                FILL fm_regex('132[0-9]{7}')
                CONSTRAINT not_null default(0)
        ::status
                TYPE SMALLINT
                FILL fm_regex('1')
                CONSTRAINT not_null default(20)
        ::timezone
                TYPE VARCHAR(32)
                FILL fm_regex('Europe/Berlin')
        ::language
                TYPE VARCHAR(12)
                FILL fm_basic()
                CONSTRAINT not_null default(100)
        ::picture
                TYPE INT
                FILL fm_basic()
                CONSTRAINT not_null default(100)
        ::init
                TYPE VARCHAR(254)
                FILL fm_basic()
                CONSTRAINT default(100)
        ::data
                TYPE INT
                FILL fm_basic()
                CONSTRAINT null(100)

TABLE:users_roles(10)
        ::uid
                TYPE BIGINT
                FILL fm_reference(users:uid)
                CONSTRAINT primary_key(26) not_null default(0)
        ::rid
                TYPE BIGINT
                FILL fm_reference(role:rid)
                CONSTRAINT primary_key(26) not_null default(0)
```

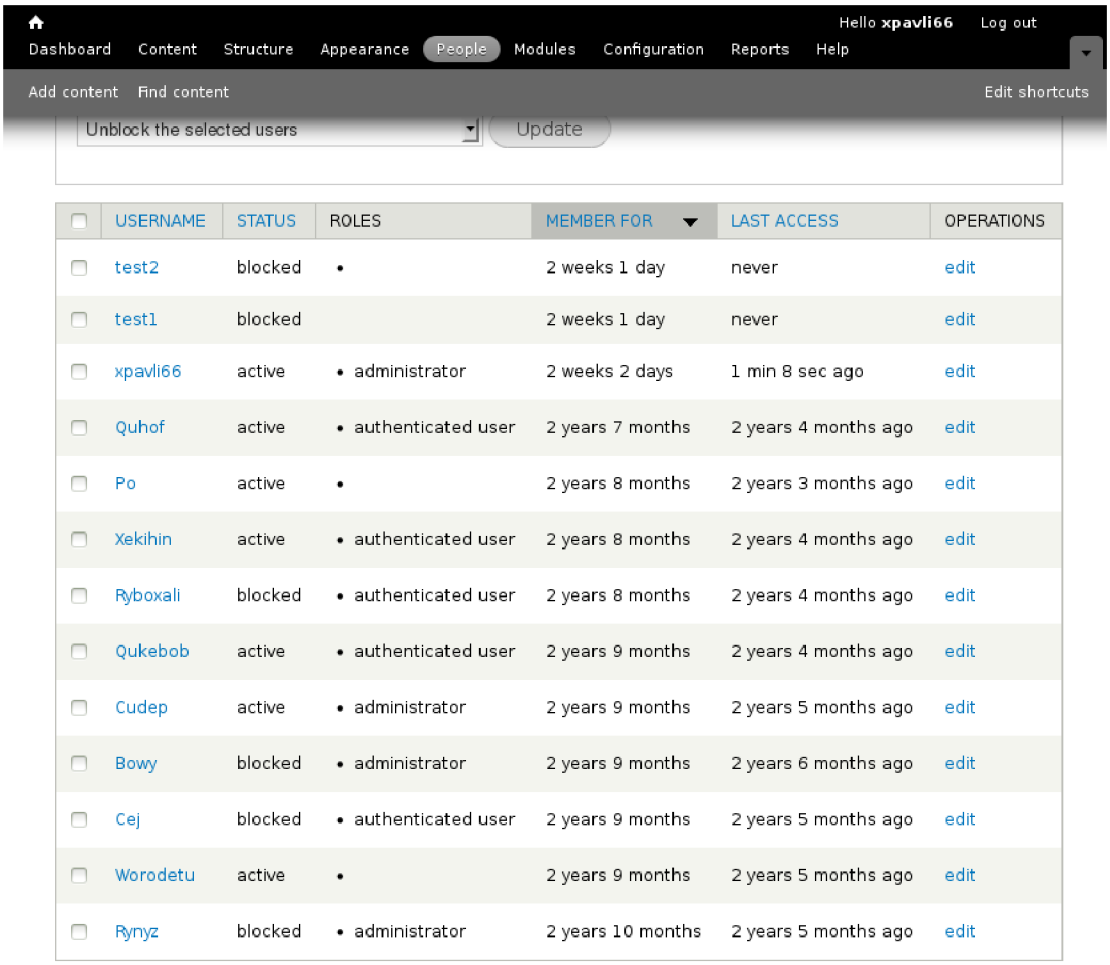
F.4 Plnění databáze

Z důvodu velikosti generovaných insertů je zde uveden pouze jeden příklad pro každou z plněných tabulek:

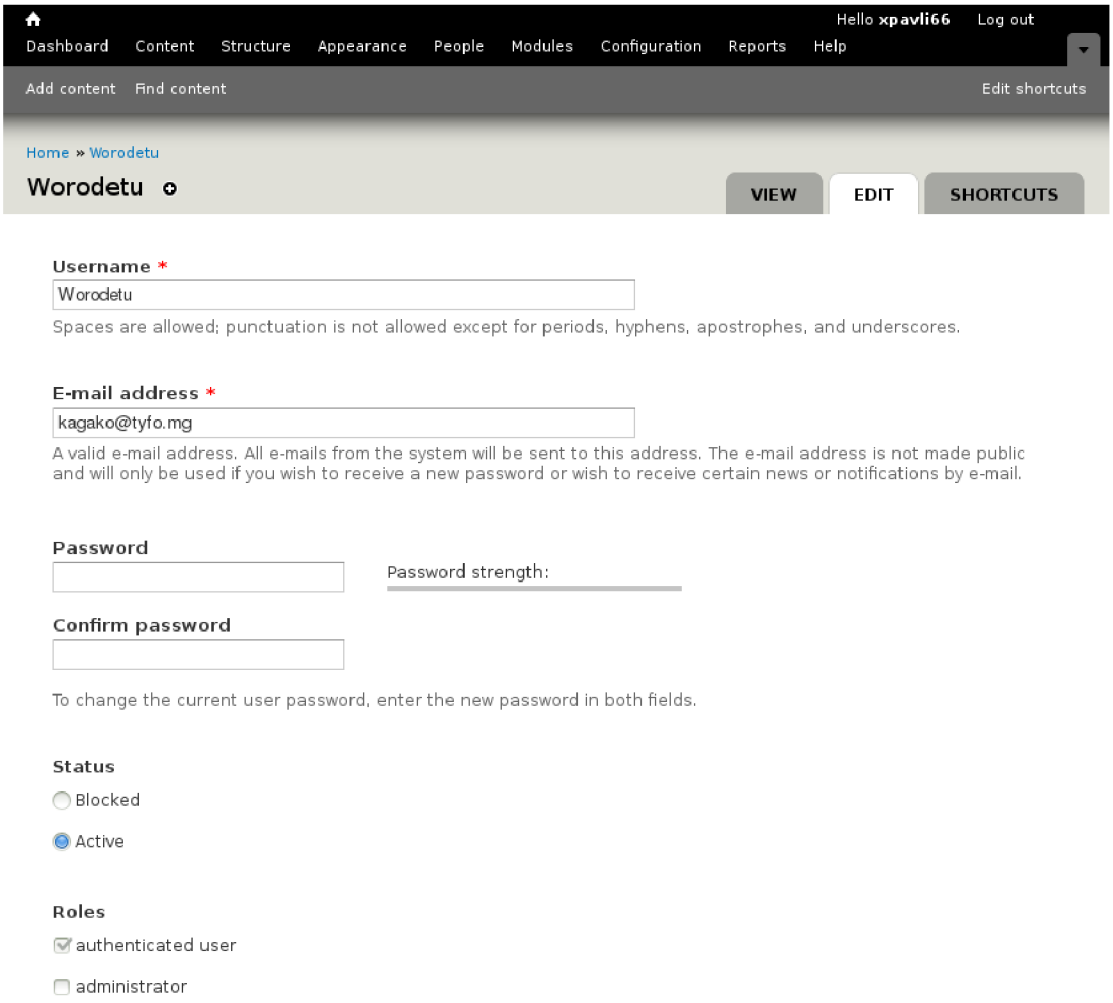
```
INSERT INTO users
VALUES (1297, 'Worodetu', 'HB$%jc9FF@', 'kagako@tyfo.mg', DEFAULT, DEFAULT,
'filtered_html', 1312008942, 1321690531, 1328617923, 1, 'Europe/Berlin',
DEFAULT, DEFAULT, DEFAULT, NULL);

INSERT INTO users_roles
VALUES (1297, (select rid from role offset random() * (select count(*)
from role) limit 1));
```

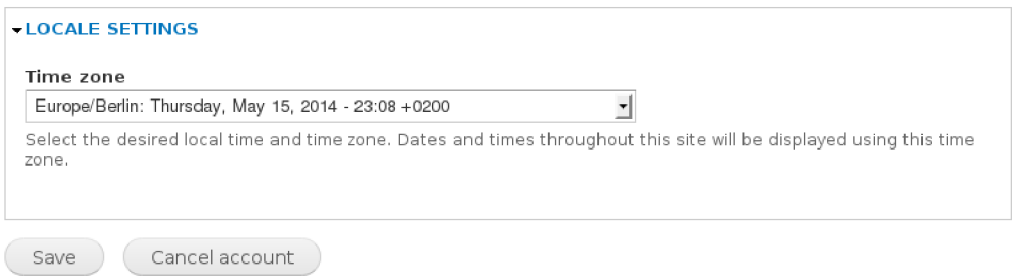
F.5 Výsledek plnění zobrazený v GUI systému Drupal



Obrázek F.1: Zobrazení vygenerovaných uživatelů



Obrázek F.2: Detaily uživatele „Worodetu“



Obrázek F.3: Časové pásmo uživatele „Worodetu“