VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

MODELEM ŘÍZENÝ VÝVOJ SPARK ÚLOH

MODEL-DRIVEN DEVELOPMENT OF SPARK TASKS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. MATÚŠ BÚTORA**
AUTHOR

VEDOUCÍ PRÁCE **RNDr. MAREK RYCHLÝ, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Bútora Matúš, Bc.**
Program: Informační technologie Obor: Informační systémy
Název: **Modelem řízený vývoj Spark úloh**
Model Driven Development of Spark Tasks
Kategorie: Paralelní a distribuované výpočty

Zadání:

1. Seznamte se s prostředím Apache Spark pro distribuované zpracování Big data a s programovacími jazyky, ve kterých ho leze použít.
2. Seznamte se s přístupem "modelem řízený vývoj" (Model Driven Development, MDD). Prozkoumejte možnosti a existující projekty pro modelování úloh zpracování dat, zaměřte se především na Big data.
3. Navrhněte nový či upravte existující grafický modelovací jazyk pro úlohy zpracování Big data v prostředí Spark. Umožněte generování zdrojového kódu Spark aplikací z jejich modelů.
4. Po konzultaci s vedoucím implementujte příslušný modelovací nástroj nebo integrujte podporu pro navržený modelovací jazyk ve stávajícím modelovacím nástroji. Ověřte funkčnost vytvořením modelů několika Spark úloh zpracování Big data a vygenerováním zdrojového kódu.
5. Výsledek zdokumentujte, zhodnoťte a zveřejněte jako open-source.

Literatura:

- Holden Karau. *Learning Spark*. First edition, 256 pp. ISBN 978-1-449-35862-4
- Databricks Spark Reference Applications. 2017. [https://databricks.gitbooks.io/databricks-spark-reference-applications]
- Michele Guerriero, Saeed Tajfar, Damian A. Tamburri, and Elisabetta Di Nitto. Towards a model-driven design tool for big data architectures. In Proceedings of the 2nd International Workshop on BIG Data Software Engineering (BIGDSE '16). ACM, USA, 2016. [http://dx.doi.org/10.1145/2896825.2896835]
- Abel Gómez, José Merseguer, Elisabetta Di Nitto, and Damian A. Tamburri. Towards a UML profile for data intensive applications. In Proceedings of the 2nd International Workshop on Quality-Aware DevOps (QUDOS 2016). ACM, USA, 2016. [http://dx.doi.org/10.1145/2945408.2945412]

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Rychlý Marek, RNDr., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 31. října 2018

Abstrakt

Cieľom diplomovej práce je popísať framework Apache Spark, jeho štruktúru a spôsob jeho funkcionality. Práca sa taktiež zaoberá problematikou modelom riadeného vývoja a modelom riadenej architektúry. Popísané sú ich výhody, nevýhody a spôsob využitia. Hlavná časť práce je zameraná na návrh modelovacieho jazyka prispôsobeného pre vytváranie úloh v prostredí Apache Spark. Predstavená je výsledná aplikácia, ktorá ponúka užívateľovi možnosť vytvoriť graf pomocou navrhnutého modelovaného jazyka. Z aplikácie je možné vygenerovať zdrojový kód modelovaného grafu v jazyku Scala.

Abstract

The aim of the master thesis is to describe Apache Spark framework, its structure and the way how Spark works. Next goal is to present topic of Model-Driven Development and Model-Drive Architecture. Define their advantages, disadvantages and way of usage. However, the main part of this text is devoted to design a model for creating tasks in Apache Spark framework. Text desribes application, that allows user to create graph based on proposed modeling language. Final application allows user to generate source code from created model.

Klíčová slova

Modelom riadený vývoj, Apache Spark, Modelovanie úloh v Apache Spark, Modelom riadená architektúra, Data driven documents, React

Keywords

Model-driven Development, Apache Spark, Modeling Apache Spark Tasks, Model-driven Architecture, Data driven documents, React

Citace

BÚTORA, Matúš. *Modelem řízený vývoj Spark úloh.* Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Marek Rychlý, Ph.D.

Modelem řízený vývoj Spark úloh

Prohlášení

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením pána RNDr. Mareka Rychleho Ph.D. a uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Matúš Bútora
22. května 2019

Poděkování

Chcel by som poďakovať vedúcemu diplomovej práce za vedenie, ochotu a odbornú pomoc počas celého procesu tvorby práce.

Obsah

Kapitola 1

Úvod

Žijeme v dobe, v ktorej sú každý deň vytvorené enormné množstvá dát. Za posledné dva roky bolo vygenerovaných 90% dát celého sveta. Samozrejme, dáta majú určitú hodnotu a s obrovským nárastom dát narastajú aj možnosti, ako ich hodnotu hľadať. Pri spracovaní veľkého množstva dát alebo dát presahujúcich klasické veľkosti, zvané Big data, tradičné metódy ich spracovania nie sú dostačujúce. Tento problém adresoval Apache Hadoop so svojím programovým modelom Map Reduce, ktorý sa zameriava na spracovanie Big data. Časom sa však ukázalo, že Map Reduce nie je úplne dostačujúcim riešením. Vtedy prišiel na rad Apache Spark postavený na novom modely abstrakcie dát riešiaci nedostatky Hadoop. Populárnym sa stal najmä vďaka svojej rýchlosti a prispôsobivosti. Framework Apache Spark je predstavený v kapitole 2.

Ďalšou aktuálnou témou dnešnej doby je modelovanie softvéru, či už z dôvodu dokumentácie, uľahčenia komunikácie medzi stranami podieľajúcimi sa na vývoji alebo uľahčení procesu samotného vývoja vďaka schopnosti generovania kódu z daného modelu. Poslednou možnosťou sa zaoberá paradigma modelom riadený vývoj a modelom riadená architektúra. Úvod do tejto problematiky, jeho výhody a nevýhody sú popísane v kapitole 3.

Motiváciou pre vznik tejto práce bolo vytvoriť aplikáciu, ktorá uľahčuje užívateľovi vytvárať úlohy vo frameworku Apache Spark a zobrazuje závislosti medzi jednotlivými príkazmi zdrojového kódu. Následne poskytuje možnosť z vytvoreného modelu vygenerovať spustiteľný zdrojový kód v jazyku Scala implementujúci úlohy vo frameworku Apache Spark.

Z možností navrhnúť vlastný alebo rozšíriť existujúci modelovací jazyk, bola vybraná možnosť vlastného návrhu. Ten popisuje, akým spôsobom je možné modelovať jednotlivé úlohy Apache Spark. Po zhotovení modelovacieho jazyka, bolo nutné implementovať nástroj, pomocou ktorého bude model v navrhnutom modelovacom jazyku vytváraný. Rozhodol som sa implementovať webovú aplikáciu, ktorá poskytuje rôzne možnosti ako proces modelovania zjednodušiť. Aplikácia pozostáva z dvoch častí — serverovej časti a grafického užívateľského rozhrania. Serverová časť dodáva funkcionalitu grafickému užívateľskému rozhraniu, v ktorom prebieha samotný proces tvorby modelu. Aplikáciu ako celok je možné využívať aj bez serverovej časti, ale s obmedzenou funkcionalitou.

Aktuálne riešenia vývoja softvéru na základe modelu a spracovanie Big data založených na modely sú rozobrané v kapitole 4. Prvé riešenie sa priamo zaoberá spracovaním Big data s využitím prístupu modelom riadeného vývoja. Druhé riešenie pristupuje zaujímavým a efektívnym spôsobom k analýze a spracovaniu veľkých dát. Posledným zmieneným je spôsob modelovania softvéru na základe matematických modelov. Hlavná časť kapitoly 5 je venovaná popisu vlastného modelu, ktorý umožňuje vytváranie úloh vo frameworku Apache Spark. Model je reprezentovaný v podobe grafu. Výsledkom je návrh

webovej aplikácie ponúkajúcej užívateľom grafické rozhranie pre interaktívne modelovanie Apache Spark úloh. Aplikácia taktiež poskytuje možnosť z vymodelovaného grafu generovať spustiteľný kód v jazyku Scala. Po predstavení návrhu aplikácie nasleduje popis samotnej implementácie. Kapitola 6 obsahuje popis návrhu a implementácie grafického užívateľského rozhrania a spôsob interakcie užívateľka s rozhraním. Nasleduje postup vytvorenia aplikačného programovateľného rozhrania (API) slúžiaceho na komunikáciu klientskej a serverovej časti aplikácie. Záver kapitoly sa venuje metóde generovania zdrojového kódu z grafu vytvoreného užívateľom a rozdielom medzi návrhom a samotnou implementáciou. Metódy testovania funkčnosti aplikácie, užívateľského rozhrania a užívateľskej skúsenosti sú zachytené v kapitole 7. Zdrojové kódy, grafy a výstupy sú zahrnuté buď priamo v tejto kapitole, alebo dodatočne v prílohách práce. Poslednou kapitolou celej práce je záver, v ktorom je zhrnutý cieľ práce spolu s dosiahnutými výsledkami. Navrhnutý je aj ďalší postup vývoja aplikácie.

Kapitola 2

Spark

Apache Spark je open-source framework vytvorený na distribuované a paralélne spracovanie dát vo výpočtových clustroch[1]. Framework ponúka množstvo API, čím zaručuje podporu populárnych programovacích jazykov ako Scala, Java, Python, R a SQL. Vytvorenie jednoduchých programov a ad hoc dátovej analýzy uľahčuje vstavaný Python a Scala príkazový riadok. Taktiež nechýba podpora pre vývoj .NET aplikácií pomocou open-source projektu od Microsoftu. Projekt s názvom Mobius[2] pridáva nie natívne podporované C# API a tým umožňuje vyvíjať Spark aplikácie v .NET frameworku [29]. Okrem vyššie zmienených API ponúka množstvo vstavaných knižníc pre pohodlný a rýchly vývoj. Hlavnou úlohou je ponúknuť používateľom jednotnú platformu pre vývoj aplikácií na spracovanie Big data. Spark je navrhnutý tak, aby poskytoval širokú škálu operácií pre dátovú analýzu. Začínajúc od jednoduchého načítania dátových súborov, SQL dotazov po machine learning, grafy a distribuované výpočty [20].

Podobne ako Hadoop, Apache Spark tiež rozširuje známy MapReduce model, ktorý je veľmi bezpečný s ohľadom na chybovú toleranciu a korektnosť výpočtov. Zaistenie týchto vlastností však prináša aj isté nevýhody a nimi je hlavne časová náročnosť distribuovaných výpočtov. Dáta sa okamžite zapisujú do HDFS (Hadoop File System), aby bola zaistená chybová tolerancia, a tým je vyžadovaný neustály presun dát z miesta ich uloženia (HDFS) do miesta výpočtu (MapReduce). Pre vyriešenie problému s neustálym presunom dát využíva Hadoop framework k manažovaniu zdrojov YARN.

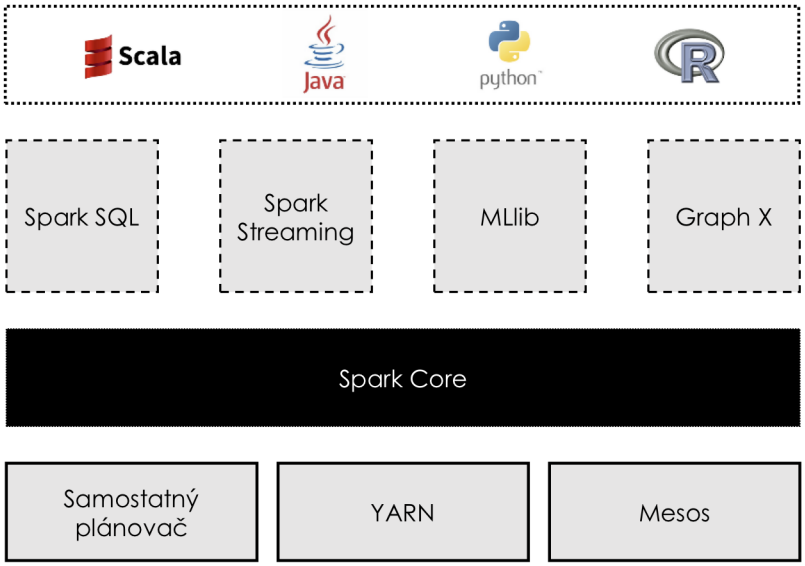
Spark sa stal populárnym hlavne vďaka svojej rýchlosti a prispôsobivosti. Rýchlosť primárne dosahuje vďaka novému modelu RDD, ktorý vznikol ako riešenie vyššie zmieneného problému s MapReduce modelom. *Resilient distributed datasets* (RDDs, pružné distribuované datasety), ktorých dáta sú uložené počas výpočtu v pamäti, čím eliminujú časovo náročné operácie zápisu na disk. Ukázalo sa však, že Spark je efektívnejší ako MapReduce aj pri komplexnejších aplikáciach využívajúcich diskové operácie [17][23]. Jeho používanie je kompatibilné s veľkou škálou perzistentných dátových úložísk, vrátane cloudov ako Microsoft Azure, Amazon S3, distribuovaných systémov ako Apache Hadoop alebo lokálnym systémom užívateľa. Prenos dát je časovo náročná operácia, preto sa Spark zameriava na vykonávanie výpočtových operácií v pamäti bez rozdielu na to, kde sú konkrétne dáta uložené [20]. Je vhodným nástrojom pre iteratívne výpočty, interaktívne dotazy a dávkové spracovanie.

[1] Počítačový cluster je zoskúpenie počítačov väčšinou prepojených počítačovou sieťou, ktoré spolu úzko spolupracujú za účelom zvýšenia výpočetnej sily.

[2] GitHub repozitár projektu — https://github.com/Microsoft/Mobius

2.1 Štruktúra

Spark ponúka rôzne knižnice a moduly, rozširujúce základnú funkcionalitu. Okrem hlavných komponentov vymenovaných a stručne popísaných nižšie, sú k dispozícií stovky ďalších open-source projektov dostupných napríklad na https://spark-packages.org/ [20]. Jednotlivé komponenty nie sú priamo integrované do jadra frameworku. Tento fakt prináša značné výhody a robí jadro veľmi flexibilným a ponúka vývojárom rôzne spôsoby prístupu k jednotlivým problémom. Jednou z výhod je, že všetky komponenty vyššej úrovne prosperujú zo zlepšení nižších úrovní, napríklad v dôsledku ich optimalizácií. Najväčšou výhodou však zostáva možnosť vývoja aplikácie spôsobom, ktorý kombinuje používanie rôznych modelov do jednom celku. Jeden používateľ môže získavať dáta z dátových tokov z rôznych uzlov naprieč clustrom, aplikovať na ne algoritmy strojového učenia, zatiaľ čo ďalší používateľ môže nad rovnakými dátami aplikovať SQL dotazy za účelom dátovej analýzy [23][17]. Štruktúra Apache Spark je zobrazená na obrázku 2.1. Uvedené sú základné komponenty spolu s najpoužívanejšími programovacími jazykmi, v ktorých je framework možné používať.



Obrázek 2.1: Štruktúru prostredia Apache Spark s jednotlivými komponentami popísanými v nasledujúcich odstavcoch. Zahrnuté sú aj najpoužívanejšie jazyky, v ktorých je framework možné využívať.

Spark Core

Zahŕňa základnú funkcionalitu frameworku, ako je napríklad správa pamäti, plánovanie úloh alebo zotavenie po chybách [23]. Implementuje základné API potrebné pre prácu s RDD, hlavnou abstrakciou dát v Sparku, ktorá je bližšie popísaná v sekcií 2.2.

Spark SQL

Spark SQL je modul určený na spracovanie štruktúrovaných dát pomocou jazykov SQL a HQL[3]. Okrem samotnej manipulácie s dátami pomocou SQL, Spark umožňuje integráciu

[3]Hive Query Language, SQL jazyk využívaný v prostredí Apache Hive.

SQL dotazov spolu s ostatnými dátovými operácia poskytovanými RDD do jednej aplikácie [23]. Abstrakcia dát na tejto úrovni je vo forme dátových rámcov, zvaných *DataFrames*. Nechýba podpora pre komunikáciu pomocou príkazového riadku a JDBC/ODBC[4].

Spark Streaming

Komponent rozširujúci základné Spark API za účelom spracovania dátových tokov v reálnom čase. Toto rozšírene API umožňuje škálovateľné, vysoko priepustné spracovanie dátových tokov s odolnosťou voči chybám, ktorých manipulácia je obdobná ako pri klasických RDD. Na úrovni Spark Streamingu sú dáta abstrahované do oddeleného toku, *discretized stream* alebo skrátene *DStream*, reprezentujúceho nepretržitý tok dát s možnosťou aplikácie funkcionality ponúkaných inými modulmi ako napríklad machine learning [21]. Interne sú reprezentované ako sekvencia viacerých RDD [14].

MLlib

ML je skratka pre machine learning, z čoho vyplýva aj názov knižnice. MLlib je knižnica obsahujúca bežné algoritmy pre podporu strojového učenia (machine learning) implementovanými ako Spark operácie nad RDD [17][24]. Ponúka nasledovné typy machine learning algoritmov iterujúcich cez veľké datasety:

- Klasifikácia
- Regresia
- Frekventované množiny (Algoritmus FP-growth)
- Odporučovací systém
- Selekcia a výber rysov
- Clustering
- Štatistika
- Lineárna algebra

GraphX

Ako už názov napovedá GraphX je knižnica pre prácu a paralélne výpočty s grafmi. Rovnako ako Spark SQL a Spark Streaming rozširuje základné API a ponúka prostriedky umožňujúce rôzne manipulácie s grafmi ako aj bežné grafové algoritmy [23].

2.2 Resilient distributed datasets (RDDs)

RDDs sú základnou abstrakciou dát v prostredí Apache Spark, ktorá zapúzdruje ich rozdelenie v clustri a zabezpečuje ich paralelizmus. Dajú sa popísať ako nemenná (read-only) distribuovaná kolekcia objektov. Každý takýto objekt je rozdelený na časti, ktoré môžu byť

[4] Java Database Connectivity (JDBC) je API programovacieho jazyka Java, ktoré definuje spôsob pripojenia klienta k databáze. Open Database Connectivity (ODBC) definuje API štandard prístupu k systému riadenia báze dát.

spracovávané v rôznych častiach clustra. RDDs môžu obsahovať akýkoľvek objekt z jazykov Scala, Python, Java alebo novovytvorený užívateľom [23]. Odolnosť voči chybám je zaručená na základe *lineage graph* (graf pôvodu) [24], pomocou ktorého je možné vždy odznova spočítať RDD a tým aj chýbajúce a poškodené časti spôsobené chybou uzlu v clustri. Ponúkajú paralélny prístup, zápis a čítanie z disku a možnosť cachovania. Existujú dva spôsoby ako vytvoriť RDDs: paralelizovaním existujúcej kolekcie v užívateľskom programe alebo načítaním datasetu z externého úložiska, ako HDFS, Hbase alebo ktorýkoľvek zdroj podporujúci Hadoop InputFormat [10]. Výpisy 2.1 a 2.2 zobrazujú spôsoby vytvorenia RDD v jazyku Scala.

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

Výpis 2.1: Vytvorenie RDD paralelizovaním [10].

```
val lines = sc.textFile("/path/to/data.txt")
```

Výpis 2.2: Vytvorenie RDD načítaním datasetu.

RDD operácie

Po vytvorení RDD má užívateľ k dispozícií dva typy operácií [23]. V nasledujúcom texte sú obe operácie stručne popísane spolu s ich rozdielmi.

Transformácie

Ako už bolo spomenuté v sekcií 2.2, RDD je *nemenná* kolekcia objektov. K ich modifikácií sa využívajú transformácie. Tieto operácie sú aplikované na už existujúci RDD a ich výsledkom je novovytvorený RDD. Príkladmi transformácií sú napríklad `filter()` alebo `union()`. Využíva sa stratégia vyhodnocovania zvaná *lazy evaluation*. To znamená, že transformácia je vyhodnotená až v prípade, ak je jej hodnota potrebná. *Lazy evaluation* zefektívňuje priebeh Spark programov. Rovnakú stratégiu vyhodnocovania výrazov používa napríklad jazyk Haskell. Štandardne pri volaní akcie je každý transformovaný RDD znovu vyhodnotený. Avšak Spark poskytuje metódy pre zachovanie perzistentného stavu RDD v pamäti pre rýchlejší opätovný prístup k nim [10].

Akcie

Druhým typom RDD operácií sú akcie. Na rozdiel od transformácií nevytvárajú nový RDD, ale naopak vracajú výslednú hodnotu do klientského programu alebo zapisujú dáta na disk. Príkladmi akcií sú `first()` a `saveAsTextFile(`*path*`)`, kde premenná *path* predstavuje cestu k programu. Akcie vynucujú vyhodnotenie RDD transformácií, nad ktorými boli zavolané [23].

2.3 Apache Spark 2.0

Postupom času svojej päť ročnej existencie prešiel Spark rozličnými zmenami. Jednou z nich bola aj zmena typu distribuovaných kolekcií. Potreba nových typov súvisela s migráciou veľkého počtu firiem z pôvodného Apache Hadoop na Apache Spark. RDD neboli úplné

vhodné pre vytváranie rozsiahlych podnikových aplikáciía preto vo verzií 1.3 Spark predstavil DataFrame. Neskôr vo verzií 1.6 bol predstavený ďalší typ distribuovaných kolekcií DataSet. Od verzie Spark 2.0 je DataFrame len alias pre stĺpec v Dataset. V jazyku Scala reprezentovaný ako `Dataset[Row]` a v jazyku Java ako `Dataset<Row>` [13][2].

DataFrame

Rovnako ako RDD, DataFrame je nemenná distribuovaná kolekcia dát. Narozdiel od RDD princíp organizácie dát je založený na podobnom koncepte aký využívajú relačné databázy. Dáta sú ukladané do stĺpcov, nad ktorými sa vykonávajú rôzne optimalizácie. DataFrame umožňuje štruktúrovať distribuovanú kolekciu dát, čím zaručuje vyššiu mieru abstrakcie a ponúkajú špecifické API na ich manipuláciu. Táto skutočnosť a značná podobnosť s SQL robí Spark prístupným širokej škále užívateľov [3]. Zdroje vytvárania DataFramu sú rôzne a to konkrétne: súbor so štruktúrovanými dátami, tabuľky v Hive[5], RDD alebo externá databáza [13].

Dataset

Dataset je najnovší typ distribuovaných kolekcií predstavený v Apache Spark vo verzií 1.6. Oproti DataFrame prichádza s výhodou typovej kontroly v čase prekladu a možnosťou funkcionálnych operácií nad kolekciou [18]. S príchodom verzie Sparku 2.0 DataFrame a Dataset API sa spojili do jedného celku a unifikovali tak možnosti dátového spracovania naprieč knižnicami. Týmto zjednotením zužuje Spark škálu konceptov potrebných na naučenie, poskytuje jednotné vysokoúrovňové a typovo-bezpečné API [3]. Dataset API zatiaľ nie je dostupné pre Python a jazyk R. V nasledujúcom texte sú uvedené hlavné výhody zjednoteného Dataset a DataFrame API:

Statické a bezpečné typovanie Syntaktická aj sémantická analýza je vykonávaná v čase prekladu. Dataset API sú vyjadrené pomocou lambda funkcií a JVM typovaných objektov. Tieto skutočnosti umožňujú detekovať chyby v čase prekladu, avšak odhaliť neexistujúci stĺpec nedokážu. Každopádne odhaliť syntaktické a sémantické chyby v dobe prekladu značne uľahčuje prácu vývojárom [3].

Vysoká úroveň abstrakcie DataFrame ako kolekcia `Dataset[Row]` vytvára štrukturovaný pohľad na predtým semi-štrukturované dáta. Umožňuje definovať triedu v programovacom jazyku, ktorou budú dáta popísané a tým pádom dostanú štrukturovanú podobu so špecifikovaným dátovým typom [3].

Jednoduché na používanie Dataset API poskytuje funkcie vyšších úrovni podporujúce napríklad agregačné operácie. Využitím značne menšieho počtu operácií a syntaxi podobnej SQL je aplikovanie agregačných operácií na dáta neporovnateľne jednoduchšie ako v prípade RDD [2][3].

Výkon a optimalizácia DataFrame a Dataset API je vybudované nad Spark SQL engine, využívajúc *Catalyst* optimalizátor [18]. Všetky relačne typované dotazy podstupujú rovnaký druh optimalizácie, vďaka čomu je výsledná aplikácia rýchlejšia a efektívnejšia. Spark kompilátor mapuje typované JVM objekty do internej reprezentácie pamäte *Tungsten*, ktorý dokáže efektívne serializovať a desereializovať JVM objekty a taktiež generovať bytecode, prináša opäť ďalšie navýšenie rýchlosti [3].

[5]Softvér používaný na manipuláciu big data pomocou SQL dotazov https://hive.apache.org/.

Zatiaľ čo RDD ponúkajú kontrolu a funkcionalitu na nižšej úrovni, DataFrame a Dataset umožňujú vlastný pohľad a štruktúru distribuovaných dát. Novšie typy distribuovaných kolekcií taktiež prinášajú vyšší level abstrakcie, špecifické operácie a hlavne neporovnateľne zvyšujú rýchlosť výslednej aplikácie.

Kapitola 3

Modelom riadený vývoj

Modelom riadený vývoj (Model-Driven development, MDD) využíva výhody grafického modelu a vopred vybudovaných komponent aplikácie za účelom poskytnúť používateľovi nástroj na tvorbu komplexnejších aplikácií. Postupom času dopyt po aplikáciach založených na modelom riadenom paradigme značne narastá.

Model je základ MDD paradigmu, definovaný ako formálna špecifikácia funkcie, štruktúry a správania systému v danom kontexte so špecifickým uhlom pohľadu. Často je reprezentovaný kombináciou nákresu a textu. Model je typicky formálne popísaný využitím napríklad jazyka UML a vhodne doplnený popisom v ľudskej reči.

Termín modelom riadený popisuje prístup k vývoju softvéru, ktorý využíva modely ako primárne zdroje dokumentácie, analýzy, dizajnu, konštrukcie, nasadenia a údržby systému [28].

3.1 Modelom riadený vývoj

Modelom riadený vývoj je metóda vývoja softvéru, v ktorej sa model stáva súčasťou samotného kódu aplikácie a nie iba súčasťou dokumentácie. Hlavná myšlienka je presunúť vývoj softvéru na vyššiu úroveň abstrakcie, ktorá je reprezentovaná modelom nezávislom na platforme. Už z názvu vyplýva, že model je základný prvok celého paradigmu. Modelom riadený vývoj je založený na dvoch základných konceptoch: abstrakcií a automatizácií. Aplikačný model softvéru je definovaný na vyššej úrovni abstrakcie a následne transformovaný do spustiteľnej aplikácie. Samotná transformácia je kľúčovým prvkom celého procesu. Štandardne je vytvorená šablóna kódu, ktorá okrem samotného kódu obsahuje aj meta-kód pre vkladanie informácií z modelu používateľa [9].

Správny postoj k modelom riadenému vývoju využíva skutočnosť, že model je možné spúšťať za behu a transformovať ho do funkčnej aplikácie interpretovaním a následným spustením modelu. Odpadá tak potreba písať kód [16]. Nasleduje výber niektorých výhod využitia MDD paradigmu.

- Uľahčuje komunikáciu medzi členmi tímu alebo zákazníkom. Model neobsahuje detaily implementácie a preto je ľahšie pochopiteľný aj pre ľudí nezainteresovaných vo vývoji konkrétnej aplikácie.
- Vyššia produktivita vývojárov je podporovaná skutočnosťou, že z modelu je možné vygenerovať kód aplikácie alebo jej komponent.
- Konzistencia jednotlivých častí modelu.

- Udržovateľnosť aplikácie je dosiahnutá na základe nezávislosti modelu na implementácií konkrétneho programovacieho jazyka.
- Schopnosť znovupoužiteľnosti modelu prispieva k šetreniu času a financií podniku.
- Prispôsobivosť - pri pridávaní funkcií do biznis logiky sa mení len časť modelu popisujúca daný segment nie celý model.

Problémy nastávajúce pri modelom riadenom vývoji softvéru sú uvedené v nasledujúcom zozname.

- Transformácia modelu z vyššej úrovne abstrakcie na model alebo kód nižšej úrovne.
- Ako využívať modely? Jedna skupina vývojárov využíva model iba ako náčrt, druhá ako návrh, zatiaľ čo MDD komunita modely považuje za programovací jazyk.
- Aká notácia a modelovací jazyk by mal byť použitý za účelom poskytnúť automatizáciu? Dôležitým krokom k dosiahnutiu MDD je nutnosť zaviesť jednotný štandard pre popis modelov.
- Nájsť vhodný kompromis medzi úrovňou abstrakcie a komplexnosti jednotlivých entít modelu [19][28].

3.2 Modelom riadená architektúra

Modelom riadená architektúra (Model-driven Architecture, MDA) je čiastočná vidina modelom riadeného vývoja navrhnutá konzorciom Object Managment Group (OMG) a preto spolieha na používanie štandardov OMG. MDA môžeme chápať ako podmnožinu MDD, ktorej modely a transformácie sú štandardizované konzorciom OMG [26].

MDA oddeľuje biznis logiku a aplikačnú logiku od technológií konkrétnej platformy. Umožňuje tak modelovať aplikáciu nezávislú na platforme popísanú UML alebo inými štandardami asociovanými s OMG. Vývoj softvéru využívajúceho MDA začína s modelom nezávislým na platforme[1] založenom na jazyku MetaObject Facility[2]. Takto vytvorený model zostáva stabilným aj v prípade vývoja technológie, v ktorej bude implementovaný výsledný kód. Využitím prostriedkov poskytovaných MDA je PIM konvertovaný do modelu špecifickej platformy[3], ktorý je následne transformovaný do spustiteľnej implementácie, napríklad Web Services, XML/SOAP, EJB, C#/.NET alebo iných. Prenositelnosť a interoperabilita sú zakomponované v architektúre [8].

PIM a PMS bývajú zväčša definované modelovacím jazykom Unified Modeling Language, čo robí OMG štandardy základom MDA.

[1]Platform-Independent Model, skrátene PIM je model nezávislý na špecifickej technológií, ktorá bude využitá na jeho implementáciu.

[2]MetaObject Facility, srkátene MOF je štandard využívaný pri modelom riadenom vývoji.

[3]Platform-Specific Model, skrátene PSM je softvérový model prepojený so špecifickou technologickou platformou. Príklad takejto platformy môže byť operačný systém alebo programovací jazyk.

Paradigma MDA je rozsiahle využívaná v rôznych odvetviach ako sú financie, biotechnológie alebo vesmírne technológie. Podľa OMG sa aplikácie založené na MDA rozdeľujú do troch skupín:

- **prenikavé služby** zahŕňajúce potreby podniku ako transakcie, bezpečnosť a obsluhu udalostí,
- **doménové objekty** používané v odvetviach ako zdravotníctvo, výroba, telekomunikácie, biotechnológie apod,
- **samotné aplikácie** vytvorené a udržiavané výrobcom alebo koncovými užívateľmi.

XML Metadata Interchange

XML Metadata Interchange (XMI) definuje formát vzájomnej výmeny metadát založený na XML. Definuje mapovanie z UML do XML. Slúži ako forma komunikácie medzi UML a ostatnými modelmi založenými na MOF. V neposlednej rade môže byť využitý na serializáciu modelov vymodelovaných pomocou iných jazykov. Efektívne štandardizuje ako sú popísané množiny metadát a umožňuje tak používateľom z rôznych odvetví jednotný pohľad na metadata. Ideálne umožňuje rôznym spolupracujúcim spoločnostiam využívať vzájomne dátové repozitáre. Ekvivalent XMI je Open Information Model od spoločnosti Microsoft [27].

Kapitola 4

Existujúce riešenia

V nasledujúcej kapitole sú uvedené niektoré z existujúcich riešení modelom riadeného vývoja, ktoré boli inšpiráciou pre vytvorenie výsledného modelu. Niektoré inšpirovali ako celok, iné ako časť danej aplikácie, napríklad spôsobom spracovania dát alebo vytvárania modelu. Okrem pozitívnych vlastností sa z uvedených riešení dajú vybrať aj tie negatívne, s ktorými je pri návrhu nutné počítať.

Prvé riešenie sa priamo zaoberá spracovaním Big data s využitím prístupu modelom riadeného vývoja. Druhé riešenie pristupuje zaujímavým a efektívnym spôsobom k analýze a spracovaniu veľkých dát. Posledným zmieneným je spôsob modelovania softvéru na základe matematických modelov.

4.1 Dataflow

Dataflow je jednotný programový model a riadená služba slúžiaca na vývoj a následného prevedenia širokého spektra metód spracovania dát. Metódy zahŕňajú *ETL*[1], dávkové spracovanie a postupné spracovanie. Dataflow ako riadená služba alokuje zdroje *on demand* (na požiadanie) za účelom minimalizovania latencie a udržania vysokej efektivity využitia [12].

Prvým krokom pri využívaní Dataflow modelu je vytvorenie *Pipeline*[2] využitím Apache Beam SDK[3]. Apache Beam na základe kódu poskytnutého užívateľom skonštruuje zreťazené spracovanie (pipeline) a vygeneruje sériu krokov, ktoré majú byť vykonané aplikáciou realizujúcu vytvorený pipeline. Takouto aplikáciou môže byť *Cloud Dataflow managed service* na platforme Google Cloud, aplikácia tretích strán alebo lokálny stroj používateľa.

Po vytvorení pipeline a umiestnení do *Cloud Dataflow managed service* sa z nej stáva špeciálny typ služby zvanej *Cloud Dataflow job*. Službu je možné monitorovať pomocou príkazového riadku alebo grafického užívateľského rozhrania. Grafické monitorovacie rozhranie zobrazuje detailné informácie a poskytuje možnosť interakcie s danou službou. Užívateľovi je ponúknutá aj grafická reprezentácia transformácií tvorených zreťazené spracovanie. Uzol grafu reprezentuje danú transformáciu vykonanú nad vstupnými dátami. Uzly sú spojené hranami, ktoré indikujú poradie vykonaných transformácií. Užívateľ je schopný modifikovať jednotlivé uzly grafu, čím modifikuje danú transformáciu a v konečnom dôsledku aj výsledný program ako celok [4].

[1] Extract, transform, load je vo všeobecnosti spôsob presunu dát z jedného alebo viacerých systémov do cieľového systému, ktorého reprezentácia dát sa líši od reprezentácie dát zdrojových systémov. ETL sa využíva napríklad v dátových skladoch.

[2] Pipeline je sekvencia krokov, ktorá načíta, transformuje a vypíše dáta.

[3] Apache Beam je jednotný pogramový model využívajúci sa pri dávkovom a prúdovom spracovaní dát.

Dataflow ponúka spôsob ako manipulovať so vstupnými dátami aplikovaním rôznych transformácií. Výsledný program sa skladá so vzájomne interagujúcich častí, ktoré je síce nutné naprogramovať, ale užívateľovi je poskytnutá aj jeho grafická reprezentácia.

4.2 Dataprep

Dataprep je služba poskytujúca možnosť vizualizácie a manipulácie so štruktúrovanými aj neštruktúrovanými dátami za účelom dáta ďalej využívať na analýzu, *data reporting*[4] alebo strojové učenie. Obrovskou výhodou je škálovatelnosť, žiadna infraštruktúra na uvedenie aplikácie do produkcie a využívanie cloud computingu. Dataprep ponúka prediktívnu dátovú transformáciu, ktorej podnety reagujú na vstupy v užívateľskom rozhraní, čím odpadá potreba používateľa písať kód. Automatická schéma, dátové typy, možnosti spájania a detekcia anomálií umožňujú používateľom preskočiť proces profilovania dát a sústrediť sa na dátovú analýzu [5].

Najskôr užívateľ vytvorí objekt *Flow*, ktorý spája a organizuje dáta, recepty a iné objekty potrebné na generovanie výsledku. V podstate sa jedná o kontajner, v ktorom sa odohráva celá logika manipulácie s dátami. Recept predstavuje poradie príkazov, podľa ktorých sú nespracované dáta na vstupe transformované. Postup využitia nástroja Dataprep môže byť nasledovný. Užívateľ nahrá rôzne typy súborov obsahujúce dáta, s ktorými chce pracovať. Pre každý typ súborového formátu vytvorí recept udávajúci kontext dátam v súbore. Vstupy zo súborov budú pomocou receptov transformované a zjednotené do výsledného súboru a opäť pomocou nového receptu sa na zjednotené dáta aplikujú potrebné transformácie [5].

Na rozdiel od Dataflow, Dataprep nevytvára graf, ale ponúka užívateľom prehľadné rozhranie na jednoduchú orientáciou v dátach rôznych dátových typov a množstvo aplikovateľných transformácií. Celé spracovanie sa odohráva v cloude bez nutnosti udržiavania a plánovania zdrojov. Apache Spark a Dataprep zdieľajú rovnaký ciel, avšak Dataprep pristupuje k transformácií dát pomocou veľmi dobre spracovaného GUI podporovaného efektívnym spracovaním dát v cloude.

4.3 Executable UML

Executable UML (xUML) je modelovací jazyk, ktorý jasne a výstižne interpretuje požiadavky modelovaného programu. Formálny zápis xUML je podmnožinou Unified Modeling Language (UML), ktorého štandard je definovaný Object Modeling Group (OMG). Napriek tomu sa jeho sémantika značne líši od klasického UML.

Návrh sa čisto zameriava na logiku a potrebné dáta modelovanej aplikácie bez nutnosti spájať model so zvoleným programovacím jazykom, pomocou ktorého bude modelovaná aplikácie implementovaná. Pomocou xUML užívateľ vytvára platformovo a implementačne nezávislí model, ktorý nikdy neobsahuje časti kódu alebo iné implementačné prvky. Naopak časti modelu sú mapované na odpovedajúce vzorky kódu. Podľa cieľovej platformy sú určené špecifické pravidlá prekladu a proces prekladu je buď plne alebo čiastočne automatizovaný. Modely tvoria plnohodnotný programovací jazyk, ktorý je plne spustiteľný a vytvorený bez nutnosti písania kódu. xUML zastáva myšlienku, že udalosti reálneho sveta sa odohrávajú naraz a nie sekvenčne. Ponecháva na užívateľovi definovať level paralelizmu modelu podľa výslednej platformy.

[4]Data reporting je proces kolekcie dát predchádzajúci dátovú analýzu.

Väčšinou sú UML modely založené na sémantike objektovo orientovaného programovania, čím sa stávajú zbytočne komplexne a vnášajú implementačné predpoklady do ich návrhov. Tieto problémy sa xUML snaží adresovať spoliehaním na základné matematické definície. Triedny model je založený na relačnej teórií, ktorá samotná je rozšírením teórie množín, funkcií a predikátovej logiky. Stavový model a komunikácia medzi udalosťami sú založené na teórií distribuovaných sietí. Akcie sú založené na analýze toku dát. Budovanie na silnom matematickom základe prináša niekoľko výhod. Pri modelovaní užívateľ benefituje z jednoduchšieho a expresívnejšieho modelovacieho jazyka ako je klasického UML. Implementáciu programu zjednodušuje nižšia úroveň komplexnosti prvkov modelu. Fakt, že matematický model je platformovo nezávislý, umožňuje generovať kód pre širokú škálu cieľových technológií.

Základným princípom xUML je schopnosť použitím úzkeho spektra symbolov a pravidiel jasne a zreteľne vyjadriť potreby modelovanej aplikácie. Zúženie spektra modelovacích symbolov vedie k odhaleniu komplexnosti, nezvyčajných javov a chybovosti logiky modelu. Pri pohľade na model musí byť jasná logika a pravidlá modelovanej aplikácie. Opak je modelovanie pomocou tradičného UML, ktoré ponúka širokú škálu komponentov a komplikovaných pravidiel. Samozrejme výsledný model je treba preložiť na platforme závislý zdrojový kód, napríklad pomocou open source xUML modelových prekladačov ako: *Pycca*, *Rosea*, *Micca* [25][7].

Kapitola 5

Návrh

V nasledujúcej kapitole je rozobraný problém modelovania úloh v prostredí Apache Spark pre spracovanie Big data. Objasňuje možnosti využitia jazyka UML na vytvorenie daného modelu, jeho výhody a nevýhody. Nasleduje popis návrhu vlastného modelovacieho jazyka, jeho adaptáciu na daný problém a následne generovanie výsledného kódu v jazyku Scala. V poslednej sekcií je uvedený návrh architektúry aplikácie spolu s použitými technológiami.

5.1 Modelovací jazyk

UML

Ako prvotné riešenie sa ponúkalo využitie modelovacieho jazyka Unified Modeling Language (UML). UML je štandardný modelovací jazyk používaný k popisu softvéru či systému. Široká škála diagramov ponúka množstvo rôznych pohľadov na modelovaný problém. Vďaka jeho vysokej popularite nie je problém nájsť vhodný nástroj na vytvorenie vybraného diagramu. Niektoré modelovacie nástroje podporujú transformáciu diagramu do textovej reprezentácie alebo priame generovanie zdrojového kódu v špecifickom programovacom jazyku. Text alebo kód by boli následne transformované do jazyka Scala ako spustiteľná aplikácia.

Prečo teda nevyužiť UML? Dôvody sú hneď dva. Jazyk UML sa javí ako príliš komplexný, čo je samozrejme jeho výhodou, ale nie príliš vhodný na popis jedného špecifického jazyka, respektíve frameworku. Ako najvhodnejší diagram pre popis typov distribuovaných kolekcií Sparku a ich transformácií sa javí kompozitný diagram. Triedny diagram je bezpochyby vhodným modelom pre popis objektov a tried, ale modelovať RDD operácie pomocou triedneho diagramu nie je vyhovujúce. V oboch prípadoch množstvo komponentov ponúkaných diagramami zostáva nevyužitých. Napriek zmieneným nedostatkom modelovať Spark úlohy pomocou UML je určite možné. Druhým a hlavým dôvodom je ponúknuť užívateľovi interaktívny nástroj, ktorý samotný proces modelovania uľahčuje.

Vlastný návrh

Ako už bolo spomenuté, dôvodom na vytvorenie vlastného modelovacie jazyka je poskytnúť užívateľovi interaktívny spôsob modelovania úloh v Apache Spark s možnosťou generovania spustiteľného kódu v jazyku Scala. Výsledný program je reprezentovaný grafom, ktorého uzly reprezentujú určitú dátovú abstrakciu frameworku Apache Spark, respektíve jazyku Scala. Hrany medzi uzlami reprezentujú vzťah medzi jednotlivými abstrakciami. Pre lepšiu orientáciu v grafe je možné jednotlivé časti patriace k danému uzlu nezobrazovať a zob-

raziť iba uzol, ktorý tieto elementy zapúzdruje. V nasledujúcej časti je uvedený spôsob modelovania jednotlivých častí programu.

Objekty a triedy

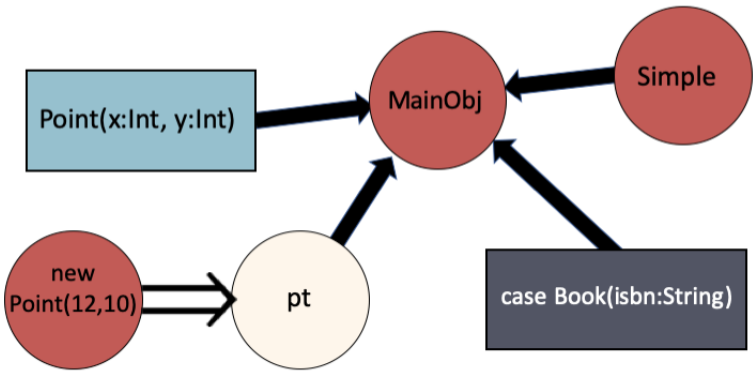
Objekt v jazyku Scala je singleton vytvorený „lenivo“, až keď je naň vyžiadaná referencia. Rovnako ako v UML je aj v tomto prípade singleton odlíšený od ostatných tried. Uzol stromu reprezentujúci objekt má tvar kruhu. Telo objektu môže obsahovať definície tried a ďalších objektov, vytvárať ich inštancie alebo ich importovať. Rovnako ako v iných programovacích jazykoch objekt môže rozširovať triedy a rysy.

Definície tried sú vizuálne zastúpené ako obdĺžniky. Vzhľad špeciálnych typov tried ako `case class` a `implicit class` zostáva rovnaký, ale pred názvom triedy v uzle stromu je potrebné uviesť príslušné kľúčové slovo. Typy tried sú v rámci modelu rozlíšené daným identifikátorom a farbou uzlu. Triedy môžu rozširovať rysy a triedy alebo vytvárať nové inštancie objektov a tried. Nová inštancia triedy je reprezentovaná rovnako ako objekt, kruhom s odlišnou farbou. Pri inštancovaní sa používa kľúčové slovo `new` rovnako ako v jazyku Scala, v prípade že ho nie je nutné uvádzať v jazyku Scala, je vynechané aj v modeli. Definícia a inštancia objektu je odlíšená farbou uzlu v grafe.

Rysy slúžia k zdieľaniu rozhrania a atribútov medzi triedami. Sú podobné rozhraniu v jazyku Java. Objekty a triedy môžu rozširovať rysy, ale nemôžu ich inštancovať. Rysy nemajú parametre a definujú sa pomocou kľúčového slova `trait`. Užitočné sú hlavne ako generické typy s abstraktnými metódami. Vizualizácia definície rysu je opäť obdĺžnik s kľúčovým slovom `trait` odlišujúci sa farbou od tried. Generický typ je uvedený v tele uzlu v hranatých zátvorkách za menom triedy a zápis je ekvivalentný ako v jazyku Scala.

Prípad, že modelovaná trieda rozširuje alebo importuje inú triedu je znázornený neorientovanou hranou reprezentovanou ako prerušovaná čiara, ktorá spája dva uzly. Nad hranou je uvedené kľúčové slovo `import` v prípade importu alebo `export` v prípade opačnom.

Samozrejme všetky zmienené entity môžu obsahovať atribúty a metódy, ktorých vizualizácia je popísaná v nasledujúcom texte.



Obrázek 5.1: Ukážka vizuálnej reprezentácie objektov a tried. Vytvorenie novej inštancie triedy `Point` a jej priradenie do premennej.

Premenné, konštanty a atribúty tried

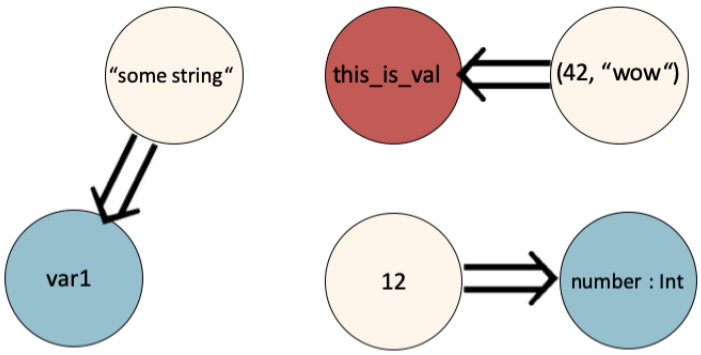
Oboje sú opäť raz reprezentované v tvare kruhu odlíšené od seba aj od vyššie zmienených farbou. Môže sa zdať mätúce reprezentovať väčšinu častí jazyka rovnakým geometrickým útvarom, ale pridelenie hrán k uzlom doplní potrebnú sémantiku a GUI poskytuje užívateľom informácie o type konkrétneho uzlu a zároveň zobrazuje informácie o spôsobe vizuálnej reprezentácie jednotlivých častí jazyka.

Premenným môžu byť priradené hodnoty všetkých dátových typov podporovaných v jazyku Scala. Dátový typ premennej je buď špecifikovaný užívateľom alebo odvodený z priradenej hodnoty. V prvom prípade za názvom premennej nasleduje znak „:“, za ktorým je uvedený dátový typ rovnako ako v jazyku Scala.
Uzol teda obsahuje hodnotu: `nazov_premennej : datovy_typ`. V prípade typovej inferencie je v uzle obsiahnutý iba názov premennej.

Model a taktiež Scala podporujú tri prístupové modifikátory: `private`, `protected` a `public`. Pri špecifikovaní modifikátoru premennej, je tento modifikátor nutne uviesť pred jej názov v uzle. Výnimka je v prípade verejného prístupového modifikátoru, ktorý je pridelený ako predvolený.

Rovnaké pravidla platia aj pre nemenné premenné (konštanty) ktoré sú v jazyku Scala deklarované pomocou kľúčového slova `val` a v modely odlíšená farbou uzlu. Špeciálny prípad nastáva v prípade `tuple`, vtedy je možné priradiť do konštanty n-ticu. Grafická reprezentácia priradenia zostáva zachovaná. Hodnota priraďovaná premennej je taktiež znázornená kruhom obsahujúcim n-ticu.

Na obrázku 5.2 je zobrazená hrana v tvare šipky s dvoma čiarami reprezentujúca priradenie hodnoty premennej. V prípade premennej `var1` je vidieť typovú inferenciu, premenná `number` ma špecifikovaný dátový typ. V prípade konštanty `this_is_val` je znázornené priradenie n-tice.



Obrázek 5.2: Typy priradenia hodnoty premennej.

Metódy a funkcie

Metódy tried a funkcie objektov bude reprezentovať šesťuholník. Rovnako ako v prípade premenných je možné špecifikovať návratový dátový typ alebo typovú inferenciu ponechať kompilátoru. V prípade špecifikácie dátového typu návratovej hodnoty funkcie sa jej

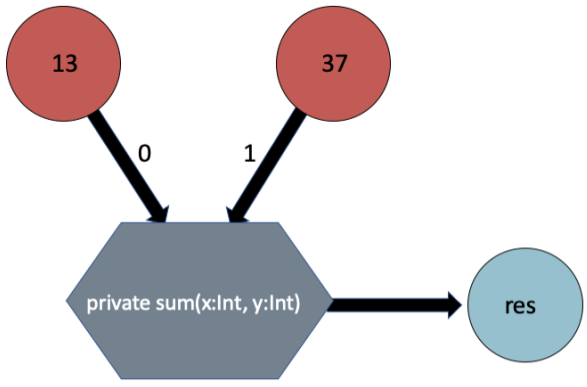
hodnota uvedie za názov funkcie v uzle grafu nasledovne: `nazov_funkcie : datovy_typ`. Rovnako ako v jazyku Scala, je funkcia bez návratovej hodnoty dátového typu `Unit`, ktorý nie je potrebné explicitne uvádzať. Prístup k modifikátorom prístupu triednych metód je ekvivalentný s prístupom popísanom pri premenných.

Vstupné parametre funkcie sú reprezentované uzlom stromu v tvare kruhu a hranou v tvare šípky smerujúcej do danej funkcie. Ich poradie je označené celým číslom začínajúc nulou. Čísla korešpondujú s poradím parametrov v deklarácií funkcie. Návratová hodnota je modelovaná ako šípka smerujúca z funkcie do uzlu predstavujúceho návratovú hodnotu (viz. podkapitola 5.3).

Telo funkcie je možné definovať vpísaním priamo do uzla. Túto metódu vytvárania funkcií je možné kombinovať so vstupnými parametrami v podobe uzlov grafu a špecifikovať tak, aké operácie budú na ne aplikované pred výstupom z funkcie. Pri vizualizácií grafu je zobrazená iba deklarácia funkcie, ale užívateľovi je umožnené jej definíciu zobraziť a prípadne upraviť. Invokácie metód triedy sú graficky znázornené ako cesta grafu od uzlu inštancie triedy po uzol reprezentujúci konkrétnu metódu. Uzly jednotlivých metód sú spojené neorientovanou hranou. V prípade invokácie prvej metódy triedy, ak nie je priradená do premennej, je hraná orientovaná a smerujúca do uzlu metódy. Potom je nutné hrane priradiť číslo reprezentujúce poradie takto vytvorenej operácie vo výslednom kóde. Tento spôsob je popísaný v sekcií 5.1 a viditeľný na obrázku 5.6. Grafická reprezentácia invokácie metód aj s príkladom je uvedená v podsekcií 5.1.

Model podporuje vytváranie anonymných funkcií. Uzol obsahujúci anonymnu funkciu obsahuje `()`, prípadne je v nich uvedený dátový typ. Takto vytvorený uzol je spojený s ďalším reprezentujúcim definíciu danej anonymnej funkcie modelovanú ekvivalentne ako klasické funkcie.

Model je navrhnutý primárne k zobrazeniu a aplikácií RDD operácií a transformácií DataFrame a Dataset, nie je optimalizovaný na vytváranie zložitých funkcií. V konečnom dôsledku vygenerovaný kód je možné upravovať a optimalizovať podľa potreby.



Obrázek 5.3: Zobrazenie modelovania vstupných parametrov a návratovej hodnoty funkcie.

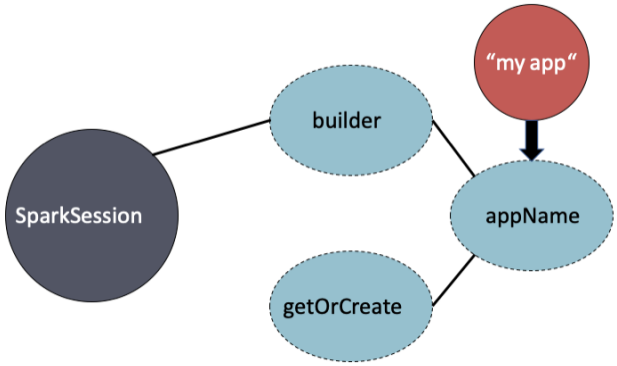
RDD, DataFrame a Dataset

RDD a ich operácie sú popísané v podkapitole 2.2 prvej kapitoly. RDD operácie bývajú spravidla na seba aplikované zreťazene, preto sú reprezentované ako za sebou idúce uzly

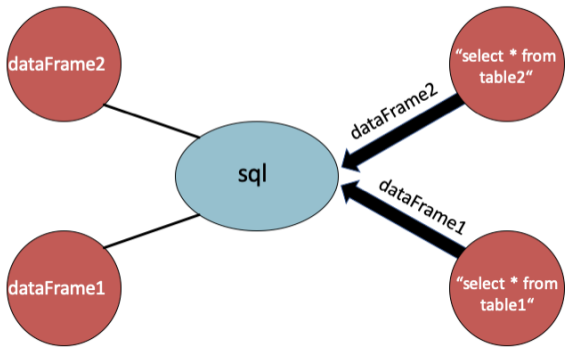
prepojené hranami. Každý uzol reprezentuje jednu RDD operáciu. Spojenie uzlu A s uzlom reprezentujúcim RDD operáciu o znamená grafickú reprezentáciu aplikovania RDD operácie o na uzol A. Vstupné parametre operácií sú modelované rovnakým spôsobom ako v prípade metód a funkcií popísaným v podkapitole 5.1 zameranej na modelovanie metód a funkcií.

Uzol reprezentujúci RDD operáciu má tvar elipsy odlišnej farby ako doposiaľ modelované uzly. Spojené sú neorientovanou hranou a zobrazené na obrázku 5.4. Uzol reprezentujúci operáciu môže byť spojený s viacerými uzlami, na ktoré bude operácia aplikovaná. V prípade ak má operácie definované vstupné parametre, budú hrany smerujúce zo vstupu do operácie označené názvom RDD, na ktorý je operácia aplikovaná. Spôsob je zobrazený na obrázku 5.5.

Spôsob modelovania DataFrame a DataSet je v podstate ekvivalentný so spôsobom popísanom v prípade RDD. Vizuálna reprezentácia zostáva rovnaká a líši sa iba farbou uzlu. Aplikácia metód na jednotlivé uzly je totožná s aplikáciou RDD operácií. SQL dotazy sú v podstate vstupom určitej metódy poskytovanou Apache Spark, preto sú modelované rovnako ako vstupné parametre funkcií.



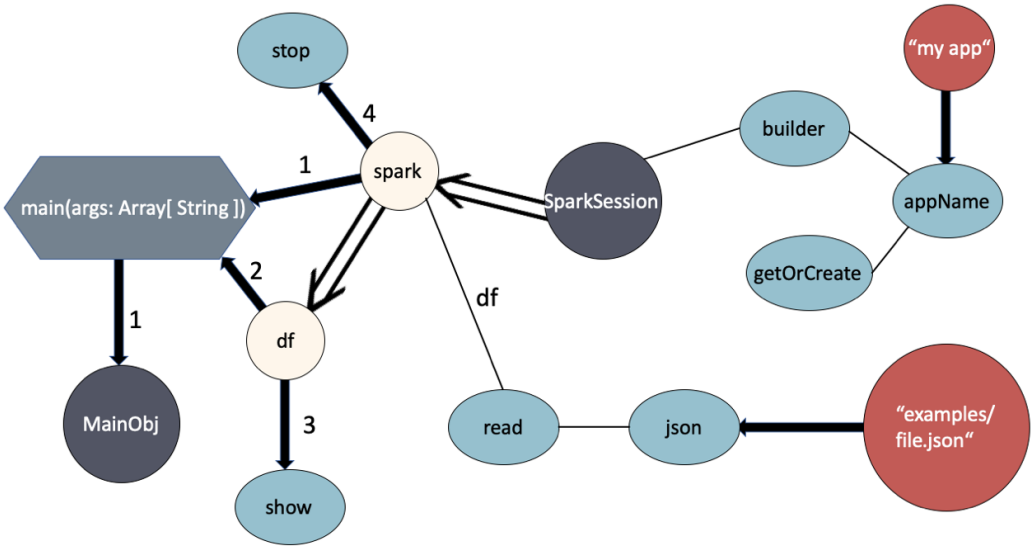
Obrázek 5.4: Invokácia metód triedy SparkSession.



Obrázek 5.5: Použitie uzlu reprezentujúceho metódu `sql` dvoma rôznymi DataFrame uložených v premenných.

Priradenie čísla hranám

Posledný krok v procese modelovania pred vygenerovaním výsledného kódu je prideľovanie čísla hranám. Poradie sa prideľuje iba hranám reprezentovanými jednoduchými šipkami. Jedná sa o hrany, ktoré predstavujú úseky kódu oddelené bodkočiarkou (semicolon) v prípade klasického programovacie jazyka, aby bolo možné na základe čísel zistiť poradie vykonávaných príkazov. Inak povedané každé číslo reprezentuje bodkočiarku. Čísla začínajú od nula pričom v každej definícií funkcie alebo objektu je vlastné počítadlo. Takto priradené čísla môžu mať rovnakú hodnotu ako čísla vstupných parametrov funkcií. Obrázok 5.6 znázorňuje vyššie popísané kroky.



Obrázek 5.6: Obrázok znázorňujúci priradenie čísel hranám.

Generovanie výsledného kódu

Uzly a hrany grafu popisujúceho program Apache Spark sú reprezentované ako objekty Document Object Model (DOM), vytvorené pomocou knižnice *D3.js* popísanej v podkapitole 5.2. Využitím funkcií ponúkaných touto knižnicou je časť DOM obsahujúca graf exportovaná do formátu JSON a prostredníctvom API poskytovaných serverom je naň odoslaný HTTP request. Server požiadavku spracuje a v prípade, že nenastala chyba vráti klientovi výsledný program v jazyku Scala implementujúci užívateľom modelované úlohy vo frameworku Apache Spark. Každý uzol grafu, okrem uzlu modelujúceho triedu, je v jazyku Scala, respektíve vo frameworku Apache Spark reprezentovaný ako typ `object`, čo je trieda s jedinou instanciou, singleton. Aplikácia RDD operácie, invokácia metódy daného objektu alebo prístup k atribútu objektu je reprezentovaná hranami grafu. Vykonaním ľubovoľnej z predchádzajúcich akcií na tento objekt má za výsledok vytvorenie nového objektu modifikovaného danou akciou. Všetky takto vytvorené objekty sú znovupoužiteľné a môžu sa navzájom využívať. Uzly, ktoré modelujú triedu, sú reprezentované typom `class`.

5.2 Návrh architektúry aplikácie

Jedná sa o webovú aplikáciu s architektúrou klient-server popísanú v nasledujúcom texte v dvoch hlavných častiach. Prvou z nich je klientská časť aplikácie, pod ktorú spadá aj samotný návrh užívateľského rozhrania (GUI). Druhá časť popisuje serverovú časť s príslušným API vytvoreným na komunikáciu medzi klientom a serverom. V oboch prípadoch sú uvedené nástroje, ktoré sa budú používať pri implementácií a výhody, ktoré poskytujú.

Klient

Grafické užívateľské rozhranie

Grafické užívateľské rozhranie umožňuje užívateľovi manipuláciu grafu modelujúceho program v prostredí Spark, popísaného v sekcií 5.1. Podporuje vytváranie uzlov grafu, úpravu ich hodnoty, ale aj vymazanie daného uzlu. Vytváranie závislostí medzi jednotlivými uzlami je zabezpečené prostredníctvom hrán medzi nimi. Podobne ako v prípade uzlov užívateľ má možnosť danú hranu pridať, vymazať alebo upraviť. Po kliknutí na uzol, reprezentujúci určitú entitu vo frameworku Apache Spark, sa zobrazí zoznam dostupných metód a atribútov danej entity. Ak je entitou práve RDD, DataFrame alebo Dataset zoznam taktiež obsahuje všetky možné transformácie a metódy. Podporované sú všetky základné moduly popísané v podkapitole 2.1 o štruktúre frameworku Apache Spark. Súčasťou GUI je aj informačný panel, v ktorom je popísané mapovanie entity Apache Spark do grafovej reprezentácie. V neposlednom rade umožňuje generovania textovej reprezentácie grafu. Poskytuje používateľovi intuitívny a pohodlný nástroj na modelovanie úloh v Apache Spark.

Tvorba grafu

Nástroj použitý na tvorbu grafu, ktorý reprezentuje model programu musí umožňovať interaktívnu prácu s grafom (pridávať, upravovať a odoberať prvky grafu), poskytovať relatívne širokú škálu štýlov aplikovateľných na uzly a hrany grafu, ale hlavne disponovať možnosťou graf transformovať do textovej reprezentácie.

Vhodný nástroj pokrývajúci vyššie zmienené požiadavky je *D3.js*. Je to JavaScript knižnica určená na manipuláciu dokumentov založených na dátach využívajúca HTML, SVG a CSS. Umožňuje naviazať dáta používateľa na Document Object Model (DOM) a následne vytvoriť graf. Veľkou výhodou je jej rýchlosť a dynamická interakcia s množstvom animácií. Selektory sú definované pomocou *W3C DOM API* podporovaným natívne v moderných prehliadačoch. Elementy uzlov DOM môžu byť vybrané rozličnými predikátmi, ktoré zahŕňajú rôzne obmedzenia, hodnoty atribútov, triedy a identifikátory. Knižnica D3 taktiež poskytuje metódy modifikujúce uzly ako sú: pridanie atribútov alebo štýlov; pridanie, vymazanie alebo zoradenie uzlov a zmenu HTML alebo textového obsahu. Priamy prístup k DOM v pozadí je tiež možný [6]. Nakoniec vytvorený dokument (graf) je knižnica D3 schopná exportovať ako formát JSON, ktorý je následne na serveri transformovaný na odpovedajúci kód aplikácie v jazyku Scala.

Server

Hlavnou úlohou serverovej časti aplikácie je transformácia textovej reprezentácie grafu obdržanej od klienta do výslednej aplikácie v jazyku Scala. Okrem transformácie grafu implementuje rozhranie pre komunikáciu s klientom (API). Prostredníctvom API klient získa atribúty a metódy, ktoré objekt reprezentujúci uzol poskytuje. Server by mal byť nezávislý

na verzii Apache Spark a umožniť užívateľovi prácu s rôznymi verziami frameworku, ktorého vývoj je pomerne intenzívny.

API

Na server sú kladené dve hlavné požiadavky, podpora frameworku Apache Spark a schopnosť implementácie API poskytovaného klientovi. Ideálnou možnosťou na implementáciu je využiť jazyk, ktorý natívne podporuje Apache Spark a štandardne sa využíva pri tvorbe webových aplikácií (Python, Java, C#[1]). Rozhodol som sa využiť už mnohokrát spomínaný jazyk Scala, ktorý natívne podporuje Spark API, samotný Spark framework je implementovaný pomocou tohto jazyka a výsledný kód je taktiež v jazyku Scala. Samozrejme Scala nie je primárne určená na vývoj webových aplikácií. Nie je to však nemožné a s pomocou vhodných prostriedkov je celý proces relatívne bez väčších obmedzení. Jedným s takýchto prostriedkov je práve *Akka*.

Akka je nástroj určený na vývoj konkurentných, distribuovaných aplikácií založených na výmene správ v programovacích jazykoch Java a Scala. Poskytuje sadu knižníc založených na HTTP, umožňujúcich vybudovať integračné vrstvy aplikácie. Ponúka rôzne druhy API či už na vyššej alebo nižšej úrovni podnecujúce väčšiu flexibilitu. Takto umožní serverovej časti aplikácie poskytovať REST API na komunikáciu s klientskou časťou [1].

[1]Spark natívne neponúka C# API. Integrácia je možná prostredníctvom knižnice Mobius od Microsoftu.

Kapitola 6

Implementácia

Nasledujúca kapitola nadväzuje a rozširuje informácie spomenuté v predchádzajúcich kapitolách. Podrobne popisuje technológie, logiku a nástroje využité pri vývoji aplikácie. Rozdelená je do po sebe idúcich logických celkov, kde každý celok adresuje konkrétny problém z pohľadu implementácie. Prvá časť je venovaná grafickému užívateľskému rozhraniu a interakcií užívateľa s aplikáciou. Predstavené je aplikačné programové rozhranie slúžiace na komunikáciu medzi klientskou časťou aplikácie a serverom. Následne je v podkapitole 6.4 popísaný postup generovania zdrojového kódu z vytvoreného grafu. V poslednej podkapitole sú zhrnuté rozdiely medzi návrhom a implementáciou programu.

6.1 Grafické užívateľské rozhranie

Grafické užívateľské rozhranie je rozdelené na dve časti. Prvá časť zobrazuje graf a prebieha v nej interakcia užívateľa s grafom. Druhá časť je bočný panel poskytujúci dodatočnú funkcionalitu aplikácie a práce s grafom. Bočný panel je podrobnejšie popísaný v podkapitole 6.1.

Tvorba grafu

Graf je tvorený prvkami SVG elementu v Document object model. Každý uzol a hrana grafu sú zaobalené v `<g>` elemente, ktorý slúži ako kontajner a používa sa na usporiadanie jednotlivých prvkov v SVG. Uzly sú reprezentované ako geometrické útvary kruh, elipsa a štvorec obsahujúce atribúty, ktoré poskytujú informácie o veľkosti a pozícií uzlu. Hrany sú vykreslené ako krivka so súradnicami v rodičovskom SVG elemente. Reprezentáciu jednotlivých uzlov a hrany ako geometrických útvarov je vidieť vo výpise 6.4. Užívateľ môže ku každému prvku grafu priradiť text, preto obsahujú jednotlivé kontajnery okrem geometrického útvaru aj položku `text` korešpondujúcu s textom uzlu alebo hrany. Kontajner zapúzdrujúci hrany navyše obsahuje ďalšiu priehľadnú krivku. Jej úlohou je zväčšiť plochu, na ktorú môže užívateľ kliknúť pri interakcií s hranou. Nepriehľadná krivka indikujúca hranu grafu pokrýva malú plochu a pokus užívateľa na ňu kliknúť môže byť obtiažny.

Užívateľ vytvára uzol grafu výberom položky reprezentujúcej typ uzlu z kontextového menu, ktoré je zobrazené po vyvolaní akcie pravého kliku na časť GUI mimo bočného panelu. Po vybraní typu je uzol vytvorený pomocou selektorov[1] poskytovaných API knižnice D3.

[1] *D3-selections* — umožňujú výber prvkov Document object model na základe vstupného reťazca (*W3C selector string)*. Pomocou nich je možné aplikovať transformácie na vybrané prvky alebo z nich získať informácie.

```
<svg>
   <g transform="translate(-1,1) scale(1)">
      <g>
         <polyline points="..." id="edge47" order="1" srcId="46" dstId="44">
         </polyline>
         <polyline points="..." style="stroke: transparent;" id="clickable-edge47">
         </polyline>
         <text dx="497.5" dy="619.5">1</text>
      </g>
      <g>
         <ellipse cx="692" cy="111" rx="50" ry="25" id="model2">
         </ellipse>
         <text dx="660" dy="111">appName</text>
      </g>
   </g>
</svg>
```

Výpis 6.1: Štruktúra grafu s jedným uzlom a hranou.

Pri vytvorení sa k uzlu priradí identifikačné číslo, typ uzlu, CSS štýly a pozícia v SVG. D3 umožňuje k DOM[2] prvkom ukladať dáta, preto nie je nutné všetky informácie o uzle udržiavať ako atribúty SVG, respektíve DOM elementu.

Typ hrany si užívateľ vyberie po kliknutí na uzol. Spôsob výberu hrany je zobrazený na obrázku 7.2. Po zvolení typu je hrana vytvorená ťahaním kurzoru. Po vyvolaní JavaScript akcie *mouseup*, spustenej po skončení interakcie užívateľa s myšou počítača, hrana zostáva vytvorená v prípade, že na rovnakej pozícií ako kurzor sa nachádza aj ďalší uzol grafu. V opačnom prípade je hrana spolu s ostatnými prvkami nachádzajúcimi sa v rovnakom kontajneri (element `<g>`) vymazaná. Vymazanie je uskutočnené pomocou funkcie `remove`, poskytnutej D3 API, aplikovanej na kontajner danej hrany. Samotné vytváranie hrany je uskutočnené opäť využitím selektorov z D3. K hrane je priradené identifikačné číslo, identifikačné čísla spájajúcich uzlov, pozíciu, typ hrany a CSS štýly. Informácie sú opäť uložené kombináciou HTML atribútov a D3 API. V grafe sa vyskytujú štyri typy vizuálne odlišných hrán, ktoré sa z pohľadu implementácie líšia výberom CSS štýlov. Štruktúra grafu je zobrazená vo výpise 6.1.

Bočný panel

Bočný panel je oddelený od grafu vytvoreného v SVG elemente. Jeho implementácia je realizovaná využitím React komponentov. Hlavným komponentom je `Layout`, ktorý obsahuje komponent `BoardSvg` zapúzdrujúci SVG element s logikou grafu. `Drawer` komponent obsahuje komponenty bočného panelu. Týmto spôsobom sú rozdelené logické funkcie aplikácie do dvoch celkov.

React je JavaScript knižnica na vytváranie užívateľských rozhraní. Využíva deklaratívne pohľady (views) pre rôzne stavy aplikácie, čím zaisťuje efektívnu aktualizáciu a vykreslenie komponentov. Aplikácia ako celok je zložená z komponent. Komponenty rozdeľuje do dvoch kategórií: triedne komponenty a funkčné komponenty. Každý komponent oddeľuje logické celky aplikácie a udržuje svoj stav mimo document object model.

[2]DOM — Document object model.

Komunikácia medzi nimi je založená na posielaní tzv. *props* smerom z rodičovského komponentu do jeho potomkov [11]. React využíva virtuálny DOM ako reprezentáciu aktuálneho stavu DOM, pomocou ktorého riadi znovu-vykreslenie jednotlivých komponentov pri ich aktualizácií. Rovnako ako D3 využíva *diff* algoritmus, nazývaný *reconciliation* algoritmus, za účelom zistiť, ktorá časť aplikácie aktualizovala svoj stav a je nutné ju znova vykresliť [22].

Každá React komponenta prechádza istými udalosťami životného cyklu, počnúc jej pripojením, aktualizovaním, až po odpojenie z document object model. Je nutné spomenúť aspoň dve z hlavných metód životného cyklu, ktoré sú kľúčovými pri integrácií knižníc React a D3. Metóda `render()` je najpoužívanejšou z metód životného cyklu a ako jediná z tejto skupiny je povinnou súčasťou každej triednej komponenty. Ako z názvu vyplýva jej úlohou je vykreslenie komponenty do užívateľského rozhrania. Deje sa tak pri pripojení komponenty do virtuálneho DOM alebo aktualizácií stavu komponenty. Druhá dôležitá metóda životného cyklu je `componentDidMount()`. Metóda je volaná po úspešnom pripojení komponenty do stromu (virtuálny DOM). Na rozdiel od predošlej metódy je v nej umožnené meniť stav komponenty a preto je vhodným miestom realizácie volanie API, ktoré následne môžu meniť vnútorný stav komponentu. V prípade výslednej aplikácie sa v tele metódy `componentDidMount()` uskutočňuje integrácia s D3. Tento prístup je podrobnejšie popísaný v sekcií 6.1.

```
import ...

class Layout extends Component {
   ...
   render() {
      return (
          <div>
             <Drawer>
               ...
             </Drawer>
             <main>
               <BoardSvg />
             </main>
          </div>
      )
   }
}
```

Výpis 6.2: Štruktúra komponentu Layout.

Dizajn užívateľského rozhrania

Bočný panel využíva princípy *Material design*. Je to vizuálny jazyk zjednocujúci moderné technológie a fyzikálne zákony z reálneho sveta, ako napríklad tieň, dopad svetla a pohyb. Užívateľ je schopný intuitívne odvodiť správanie jednotlivých prvkov aplikácie. *Material design* sa osvedčil napríklad v operačnom systéme *Android* a prináša používateľovi kontrolu nad moderne vyzerajúcimi prvkami užívateľského rozhrania, ktoré presne indikujú svoju funkcionalitu [15]. Aplikácia využíva knižnicu *Material-UI*, ktorá poskytuje React komponenty implementujúce Material design.

Integrácia React a D3

React a D3 zdieľajú rovnaký princíp uľahčenia práce užívateľa tým, že manipulujú document object model a jeho komplexnosť využívaním rôznych optimalizácií. Taktiež uprednostňujú „čisté funkcie“ - kód, ktorý pre daný vstup vždy vráti rovnaký výstup bez vedľajších efektov [22].

Pri integrácií oboch knižníc v jednej aplikácií nastávajú určité problémy, keďže obe zdieľajú rovnaký princíp manipulácie s document object model. Existuje viacero riešení, pričom každé prináša isté výhody a nevýhody. Výber správneho riešenia zaleží od situácie a účelu celkovej aplikácie, pričom všetky riešenia zdieľajú jednu podmienku a tou je: React a D3, by nikdy nemali zdieľať kontrolu nad DOM [22]. V princípe sa jedná o to, ktorá knižnica bude kontrolovať DOM. Existuje niekoľko možností ako pristúpiť k ich integrácií:

- D3 riadi prevažnú časť DOM a React vykresľuje SVG element,
- využitie knižníc tretích strán,
- React riadi DOM a funkcionalita D3 pozostáva z využívania matematických funkcií poskytovaných touto knižnicou — samozrejme tento spôsob zahŕňa iba matematické funkcie, ktoré nemanipulujú s document object model,
- plne využitie funkcionality API poskytovaného D3, zaobalené v metóde životného cyklu komponentu `componentDidMount` knižnice React.

Aplikácia využíva posledný zmienený prístup z vyššie uvedených možností integrácie. Tento prístup umožňuje plné využitie funkcionality poskytovanej D3, keďže má pod kontrolou podstatnú časť DOM. React slúži na vytvorenie prázdneho `<svg />` elementu, ktorý funguje ako koreňový prvok pre D3. Veľkou výhodou tohoto prístupu je stanovenie pevnej hranice medzi knižnicou D3 a knižnicou React, ktorá ak bude dodržaná vylúči nepredvídateľné správanie aplikácie. Rovnako umožňuje jednoduché prepojenie už existujúcej D3 aplikácie s knižnicou React. Naopak zdrojový kód prichádza o štruktúru navrhnutú pre React aplikácie a zahŕňa mnoho závislostí.

6.2 Interakcia s grafickým užívateľským rozhraním

Knižnica D3 riadiaca vykreslenie a funkcionalitu grafu poskytuje vlastné udalosti (JavaScript events) rozširujúce klasické JavaScript udalosti, na ktorých je založená interakcie užívateľa s grafom. V tejto sekcií sú uvedené základné spôsoby interakcie a spôsob ich implementácie.

Atribúty jednotlivých uzlov grafu sa líšia podľa geometrických útvarov, respektíve SVG elementov, ktoré graficky reprezentujú konkrétny uzol. Z tohto dôvodu nie je možné definovať jednotný spôsob manipulácie s uzlami. Reprezentácia uzlov ako SVG elementy je zobrazená na obrázku 6.4.

Ťahanie uzlu

Ťahanie uzlu využíva *d3.drag()* API aplikované na výber D3 prvku, ktoré umožňuje vybraný prvok ťahať. Umožňuje definovať funkcie vykonané pred, počas a po ukončení ťahania prvku. Samotné ťahanie je implementované tak, že ťahanému uzlu sú zmenené súradnice x a y na aktuálne súradnice posúvaného kurzoru. Výhodou je namiesto klasickej JavaScript

```
class BoardSvg extends Component {
   constructor(props) {
      super(props)
      this.mainSvg = null
   }

   componentDidMount() {
       this.mainSvg = d3.select(this.svg)
   }

   render() {
      return (
          <svg ref={svg => this.svg = svg} />
      )
   }
}
```

Výpis 6.3: Integrácia knižníc D3 a React.

```
<circle cx="145" cy="135" r="50" />
<rect x="350" y="260" width="100" height="100" />
<ellipse cx="250" cy="210" rx="50" ry="25" />
<polyline points="295, 565 217.5,547.5 140,530"></polyline>
```

Výpis 6.4: Skrátená reprezentácia uzlov a hrany grafu ako prvky SVG.

udalosti použitý *d3.event* poskytovaný D3 API. Táto udalosť používa súradnicový systém ťahaného prvku a automaticky počíta s odchylkou SVG elementu v tele HTML dokumentu. Ukážka 6.5 zobrazuje implementáciu ťahania kruhu. Elipsa využíva totožnú implementáciu a implementácia pre štvorec sa líši v atribútoch *d3.select()* vo funkcií `dragging`. Spolu s daným uzlom sa posúvajú aj s ním spojené hrany a text obsiahnutý v uzle.

Zmena veľkosti uzlu

Pri zmene veľkosti uzlu sa opäť využíva *d3.drag()* API. Po kliknutí na uzol grafu sa objaví rukovať na vybranom uzle, za ktorú užívateľ ťahá čím mení veľkosť vybraného uzlu. Jednotlivé funkcie využívané k výpočtu zmeny veľkosti sú relatívne obsiahle, preto sú uvedené len matematické vzorce hlavného výpočtu.

Rovnica 6.1 zobrazuje výpočet polomeru kruhu, ktorý je reprezentovaný premennou *radius'*. Pri zmene veľkosti kruhu pôvodné súradnice x a y zostávajú nezmenené. Mení sa iba hodnota polomeru s využitím *d3.event* súradníc popísaných v podkapitole 6.2 a pôvodných súradníc kruhu.

$$radius' = \sqrt{(x - d3.event.x)^2 + (y - d3.event.y)^2} \tag{6.1}$$

Štvorec je možné zväčšovať a zmenšovať z pravého horného rohu. Rovnica výpočtu 6.2 zahŕňa zmenu výšky (*width'*), šírky (*height'*) a novú súradnicu y (y'). Pri posúvaní kurzoru do pravého horného rohu sa súradnica y zmenšuje, ale veľkosť štvorca je potrebné zväčšiť. Z tohto dôvodu je nutné vypočítať novú súradnicu y' a zahrnúť ju do výpočtu. Premenné

```
d3.select('circle')
   .call(d3.drag()
      on('start', dragStarted)
      on('drag', dragging)
      on('end', dragEnd)
   )

function dragging(d) {
    const newPosX = d3.event.x
    const newPosY = d3.event.y

    d3.select(this)
        .attr("cx", d.x = newPosX)
        .attr("cy", d.y = newPosY)

   centerTextLabel(d3.select(this.parentNode))
   moveEdge(d.id, newPosX, newPosY)
}
```

Výpis 6.5: Ukážka implementácie ťahania uzlu.

bez indikátoru ’ reprezentujú pôvodne hodnoty výšky, šírky a súradníc x, y.

$$\begin{aligned} width' &= width + (d3.event.x - x) \\ height' &= height + (y - d3.event.y) \\ y' &= y + height - height' \end{aligned} \tag{6.2}$$

Posledná uvedená rovnica 6.3 zobrazuje výpočet zmeny veľkosti elipsy. Pri elipse na rozdiel od kruhu, obsahujúci iba horizontálny polomer, je nutné uviesť aj vertikálny polomer. Vzorec výpočtu horizontálneho polomeru (*radiusX’*) je totožný so vzorcom výpočtu polomeru kruhu uvedeného v rovnici 6.1. Vertikálny polomer (*radiusY’*) pri manipulácií zachováva pomer 1:2 s horizontálnym polomerom, preto je jeho výpočet závislí na novo vypočítanom horizontálnom polomere.

$$\begin{aligned} radiusX' &= \sqrt{(x - d3.event.x)^2 + (y - d3.event.y)^2} \\ radiusY' &= \frac{radiusX'}{2} \end{aligned} \tag{6.3}$$

Implementácia oddialenia, priblíženia a pohybu scény vykreslujúcej graf využíva funkciu *d3.zoom()* poskytovanú D3 API. Vo výsledku funkcia aplikuje CSS vlastnosť `transform`, s hodnotami `translate()` a `scale()`, na kontajner `g`, ktorý je priamym potomkov SVG elementu. Túto skutočnosť je možné vidieť na obrázku 6.1. Funkcia `scale()` umožňuje priblížiť a oddialiť scénu. Obmedzená je na hodnoty z intervalu $< 0.5, 2 >$. Pomocou funkcie `translate()` je implementovaný posun scény. Tentokrát funkcia nie je obmedzená žiadnymi hodnotami.

6.3 API

Backend aplikácie je implementovaný v jazyku Scala s využitím knižnice *Akka Http*. Pomocou nej je vytvorené REST API, slúžiace na komunikáciu s klientskou časťou aplikácie.

HTTP metóda	URL	popis
GET	/version	aktuálna verzia Apache Spark
GET	/newMethods?from=ObjectName	metódy modelovaného objektu, ktorý má zatiaľ neznámy dátový typ
GET	/knownMethods?from=ObjectName	metódy modelovaného objektu, ktorý má uložený svoj dátový typ
GET	/imports	zoznam importovaných balíčkov využívaných v aplikácií
POST	/imports	aktualizovať zoznam balíčkov využívaných v aplikácií

Tabulka 6.1: Koncové body API.

Zmena oproti návrhu nastáva v skutočnosti, že výsledný zdrojový kód v jazyku Scala je generovaný v klientskej časti aplikácie namiesto pôvodne plánovaného generovania na serveri. Dôvodom tejto zmeny bolo uvedomenie si skutočnosti, že klient obsahuje všetky informácie potrebné pre vygenerovanie kódu. Odpadá teda potreba posielať informácie o uzloch grafu a ich prepojeniach na server, kde ich server spracuje a vygeneruje zdrojový kód Spark aplikácie zodpovedajúci modelovanému grafu. Kód je nutné uložiť do súboru a opäť poslať klientovi. Celkovo tak odpadá potreba dvoch volaní REST API, avšak za cenu využitia výpočetnej sily klientského počítača, respektíve prehliadača. Každopádne nejedná sa o výpočty, ktoré by v značnej miere zaťažovali zariadenie klienta, a preto je zmena prístupu ku generovaniu výsledného kódu prípustná.

Serverová časť aplikácie sa skladá z rôznych objektov a *traits*. Traits sú podobné rozhraniam v programovacom jazyku Java a umožňujú zdieľať rozhranie, metódy a atribúty medzi triedami a objektami rožširujúcimi daný trait.

Hlavným objektom je `Server`, ktorý zapúzdruje celú serverovú časť aplikácie. Spustený je na adrese `http://localhost:8001`. Využíva trait `SparkServices` implementujúci koncové body REST API, popísané v tabuľke 6.1. Ďalšou dôležitou časťou aplikácie je objekt `AvailableImports` spravujúci využívané balíčky tried v grafe. Používateľ v GUI aplikácie zadá balíčky, ktoré chce využívať pri modelovaní. Tie sú odoslané na server a uložené pre interné potreby serverovej časti aplikácie. Používateľ je schopný aktualizovať zoznam balíčkov počas celej doby práce s aplikáciou.

Poslednou časťou je trait `Reflection`. Ten na základe volania REST API od klienta získava názov objektu. Podľa názvu objektu vyhľadá v balíčkoch, uložených v `AvailableImports`, plne kvalifikované meno daného objektu. Následne využije plne kvalifikované meno a natívne zabudovaného *Scala Reflection API* v jazyku Scala k získaniu informácií o metódach poskytovaných vyhľadaným objektom. Reakciou na klientovo volanie je zoznam získaných metód vo formáte JSON.

6.4 Generovanie výsledného kódu

Nasledujúca sekcia popisuje hlavné myšlienky algoritmu, pomocou ktorého sa z modelovaného grafu stáva výsledný zdrojový kód. Generovanie sa dá rozdeliť do troch častí: transformovať grafickú reprezentáciu do dátovej štruktúry, vytvorenú dátovú štruktúru interpretovať ako kód Spark aplikácie a v poslednej rade zápis zdrojového kódu do súboru v klientskej časti aplikácie.

Jednotlivé prvky grafu predstavujú HTML elementy, respektíve SVG elementy. Pomocou selektorov knižnice D3, sú z grafu vybrané všetky potrebné informácie o hranách

a uzloch. Informáciami sú napríklad typ, identifikačné číslo uzlu a pri hranách naviac ešte spájané uzly.

Z hrán sú následne identifikované hlavné uzly grafu, ktoré predstavujú definíciu tried a objektov. Takéto uzly sú buď spojené orientovanými hranami, alebo nie sú spojené so žiadnymi inými uzlami. V prípade orientovanej hrany, smer hrany vždy smeruje do uzlu. Neexistujú žiadne hrany, ktoré smerujú z hlavného uzlu. Ak sú splnené vyššie uvedené podmienky, je uzol identifikovaný ako definícia triedy alebo objektu, ktoré budú neskôr využité pri generovaní kódu.

Informácie o uzloch a hranách získaných z grafu využíva trieda `ExportModel` a `ChildInfo`. Prvá trieda z nich extrahuje *id*, *typ*, *text* a *pole následníkov* jednotlivých modelov. Pole následníkov je pole tried `ChildInfo`, kde každá položka pola, respektíve tejto triedy, má atribúty *id*, *id hrany*, *typ hrany* a *poradie hrany*[3]. Vyššie uvedené triedy pozostávajú z jedinej metódy, ktorou je konštruktor a slúžia ako kontajner na uloženie potrebných informácií o danom uzle. Všetky spomenuté atribúty sú získane na základe interakcie užívateľa s grafom.

Trieda `AllModels`, vytvorená podľa návrhové vzoru *Singleton* ukladá jednotlivé inštancie triedy `ExportModel` do poľa. Poskytuje metódy na uľahčenie práce s modelmi, akými sú vyhľadanie modelu podľa identifikačného čísla, zoradenie modelov podľa atribútu *poradie hrany* a poskytuje prístup k všetkým vytvoreným inštanciám `ExportModel` potrebných počas generovania kódu.

Generovanie kódu je založené na rekurzívnom volaní funkcií, ktoré prechádzajú jednotlivé modely uložené ako inštancie `ExportModel`. Pri priechode uzlu je najskôr prejdený jeden jeho následník do hĺbky, až kým tento následník neobsahuje žiadne iné uzly. Potom je postup opakovaný pre ďalších následníkov uzlu. Vymodelovaný graf obsahuje neorientované hrany a cyklus, ktoré môžu viesť k zacykleniu algoritmu alebo nesprávnemu určeniu vzťahu predchodca — následník medzi jednotlivými modelmi. Situácia hlavne nastáva pri neorientovaných hranách reprezentujúcich reťazenie RDD operácií, keďže hrany neobsahujú šípku indikujúcu smer. Problém rieši algoritmus, ktorý prechádza neorientované hrany a uzly reprezentujúce RDD operácie. Algoritmus pracuje nasledovne:

- najskôr vyhľadá všetky RDD operácie, ktoré sú listami grafu,
- ako aktuálny uzol sa nastaví listový uzol,
- z aktuálneho uzlu je vyhľadaná hrana reprezentujúca reťazenie operácií spájajúca ďalší uzol RDD operácie,
- tento novo-nájdený uzol je označený ako predchodca aktuálneho uzlu,
- následne listový uzol odstráni z poľa následníkov referenciu na predchodcu,
- aktuálny uzol sa presunie na predchodcu a postup sa opakuje pokiaľ nie je aktuálnym uzlom objekt invokujúci RDD operácie.

Samotné generovanie kódu okrem vyššie zmienených tried využíva ešte dve nové. Prvá trieda `Line` reprezentuje riadok vo výslednom zdrojovom kóde. Obsahuje atribúty *content* — v ňom je uložená textová reprezentácia vygenerovaného kódu a *number* — poradie vygenerovaného textu v oblasti pôsobenia, napríklad v tele funkcie. Jednotlivé inštancie `Line` sú uložené v triede `Scope`, ktorá definuje oblasť pôsobenia funkcií, objektov a tried. Táto

[3]Reprezentuje poradie programových častí v príslušnej oblasti pôsobnosti.

trieda má okrem riadkov aj atribúty *start* a *end* uchovávajúce text oblasti pôsobenia potrebný v zdrojovom kóde pri definovaní začiatku a konca oblasti. O správnosť a doplnenie syntaxe k textu získaného z tela uzlu sa starajú samostatné funkcie, ktoré nie sú súčasťou vyššie uvedených tried. Funkcia `scopesToString` transformuje obsah uložený v atribútoch triedy `Scope` na dátový typ reťazec. Výsledný zdrojový kód vznikne konkatenáciou pridaní tried (`imports`) a výsledku funkcie `scopesToString`.

Z bezpečnostných dôvodov webové aplikácie spustené u klienta nedovoľuje priamy zápis do súboru alebo vytvorenie súboru nachádzajúceho sa na zariadení klienta. Štandardným spôsobom ako získať súbor na zariadenie klienta je poskytnúť odkaz na súbor, ktorý si užívateľ stiahne. Výsledný vygenerovaný kód je teda vytvorený ako *Blob*[4] obsahujúci vytvorený reťazec. Poskytnutý je užívateľovi prostredníctvom odkazu na stiahnutie.

Dodatočné funkcie aplikácie

Aplikácia okrem doposiaľ spomenutých možností manipulácie s uzlami a hranami grafu umožňuje dve ďalšie funkcie. Po kliknutí na uzol grafu si užívateľ z kontextového menu môže zvoliť možnosť *preview* alebo *collapse/expand*.

Preview zobrazí modálne okno, ktorého obsahom je vygenerovaný kód. Nezobrazuje však kód celého grafu, ale iba časť obsahujúcu vybraný uzol. Hlavnou úlohou tejto funkcie je uľahčiť užívateľovi orientáciu v grafe. Implementácia *preview* využíva triedy a funkcie spomenuté v sekcií 6.4 so zameraním na konkrétny uzol namiesto celého grafu. Ukážka je uvedená na obrázku 7.3 v kapitole 7.

Collapse/expand zdieľa rovnaký cieľ ako *preview*, a to uľahčenie orientácie v grafe. Ponúka užívateľovi schovať alebo expandovať časti grafu. Po kliknutí na uzol grafu sa všetky uzly, ktoré sú následníkmi zvoleného uzlu nezobrazujú (*collapse*) alebo zobrazujú (*expand*).

Undo/redo operácie umožňujú užívateľovi zvrátiť vykonané akcie. Implementované sú pomocou triedy `UndoRedo`, vytvorenej podľa návrhového vzoru *Singleton*. Trieda udržuje pole udalostí vykonaných užívateľom. Pole je obmedzené na veľkosť desiatich udalostí. V prípade, že je pridaná nová udalosť a v poli je aktuálne uložených desať udalostí, je prvá položka pola (časovo najstaršia udalosť) vymazaná. V aplikácií nastáva šesť typov akcií:

- vytvorenie hrany,
- vymazanie hrany
- vytvorenie uzlu,
- zmena veľkosti uzlu,
- zmena pozície uzlu,
- vymazanie uzlu.

Po úspešnom prevedení každej z vyššie uvedených akcií, je konkrétna akcia pridaná na koniec pola udalostí s identifikátorom akcie a informáciami o aký prvok grafu sa jednalo. V prípade *undo* sú položky z pola postupne vyberané a podľa typu akcie je zvolený spôsob jej spracovania.

Operácia *redo* je implementovaná obdobne. Pri každej invokácií metódy `undo()`, implementujúcu operáciu *undo*, je pred spracovaním konkrétna udalosť uložená do ďalšieho pola.

[4]Blob(Binary larger object) reprezentuje objekt podobný súboru obsahujúci nemenné prvotné dáta. Reprezentuje dáta, ktoré nemusia byť nutne vo formáte natívne podporovaným v JavaScript.

Toto pole slúži na uchovávanie akcií, vykonaných operáciou *undo*, ktoré môžu byť následne zvrátené metódou `redo()`, implementujúcu operáciu *redo*. Spracovanie týchto udalostí prebieha obdobným spôsobom ako pri operácií *undo*.

6.5 Rozdiely medzi návrhom a implementáciou

Výsledná aplikácia bola implementovaná podľa návrhu predstaveného v kapitole 5. Nie je s ním však totožná, a preto v tejto časti budú spomenuté rozdiely návrhu a implementácie. Väčšinou sa jedná o vizuálne zmeny, ktoré neovplyvňujú funkčnosť aplikácie.

Oproti ukážke 5.1 pri vytvorení novej inštancie triedy nie je potrebné uvádzať kľúčové slovo `new`. Taktiež inštancia a definícia triedy a objektu nie sú odlíšené farbou.

V prípade funkcie sa tvar uzlu zmenil na elipsu oproti pôvodnému šesťuholníku. Uzol indikujúci výstupnú hodnotu funkcie je spojení s uzlom tela funkcie rovnakým spôsobom ako vstupy funkcie, líši sa však pridaním kľúčového slova `return`.

Hrana grafu reprezentujúca priradenie do premennej je vizualizovaná prerušovanou čiarou. Konštanta v grafe je vytvorená pomocou pridania kľúčového slova `val` pred názov konštanty, rovnakým spôsobom ako v prípade triedy `case`. Obrázok 5.5 zobrazuje zdieľanie RDD operácie `sql`, ktoré vo výslednej aplikácie kvôli prehľadnosti nebolo implementované. Rozdiely je možné vidieť v prílohách A.2 a A.3. Presný popis uzlov a hrán grafu je obsiahnutý v dokumentácií v archíve na CD pribalenom k textu práce.

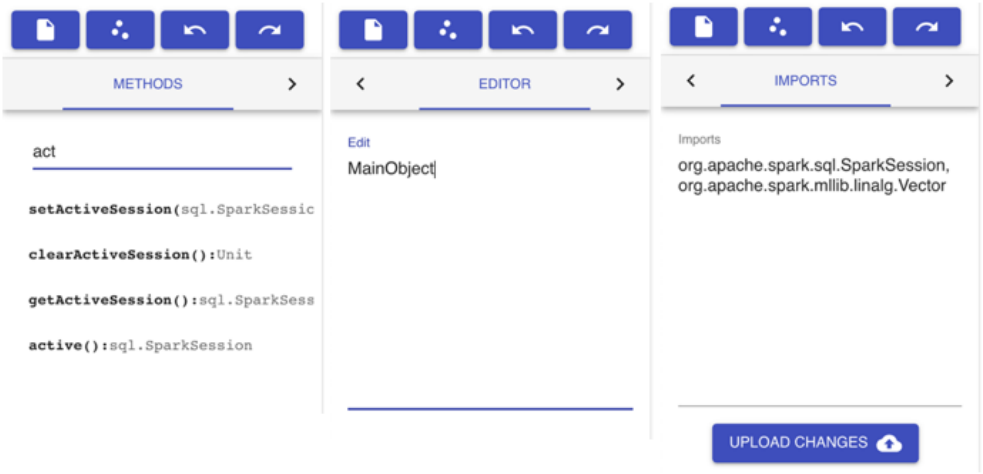
Kapitola 7

Testovanie

Nasledujúca kapitola je rozdelená do dvoch hlavných častí. Prvá časť sa zaoberá testovaním návrhu grafického užívateľského rozhrania a užívateľskej skúsenosti. Druhá časť je zameraná na testovanie generovania výsledného kódu z vytvoreného grafického modelu reprezentujúceho Apache Spark aplikáciu.

7.1 Testovanie grafického užívateľského rozhrania

Podstatná časť návrhu a testovania prebehla v rámci predmetu *UXIa: Uživatelská zkušenost a návrh rozhraní a služeb (v angličtině)*. Na začiatku semestra bol predložený *mock-up* zobrazený na obrázku v prílohe A.1, ktorého aktualizácia bola reakciou na pripomienky získané od študentov daného predmetu počas celého semestra. Oproti predbežnému návrhu, bolo nutné pridať viacero funkčných prvkov do ovládacieho panelu na bočnej strane. Ovládací panel bol rozdelený na dve časti. Prvá časť, obsahuje štyri tlačítka reprezentujúce hlavnú alebo často používanú funkcionalitu aplikácie. Druhá časť je rozdelená na záložky, medzi ktorými je používateľ schopný prepínať rôzne funkcie bočného panelu. Metóda rozdelenia jednotlivých funkcií aplikácie medzi záložky bola zvolená z dôvodu minimalizovať rozmery bočného panelu a poskytnúť prevažnú časť GUI komponentu manipulujúceho s grafom. Výsledné spracovanie je zobrazené na obrázku 7.1. Kľúčovým prvok počas aktualizácií, bolo uvedomiť si, kto sú koncoví používatelia aplikácie a ako vytvoriť jednoduché a intuitívne GUI poskytujúce používateľom čo najlepšiu užívateľskú skúsenosť.



Obrázek 7.1: Ukážka ovládacieho panelu obsahujúceho záložky. Obrázok je rozdelený do troch častí zobrazujúcich rozdielne aktuálne zobrazené záložky.

Ďalšou dôležitou súčasťou testovania bol spôsob vytvorenia hrán medzi uzlami grafu. Testovanie tentokrát neprebehlo v rámci predmetu *UXIa*, ale medzi troma na sebe nezávislými testovacími užívateľmi. Užívateľom boli poskytnuté dva spôsoby ako vytvoriť hranu medzi uzlami. Oba spôsoby predchádza udalosť kliknutia na konkrétny uzol grafu, ktorý používateľ chce spojiť s iným uzlom.

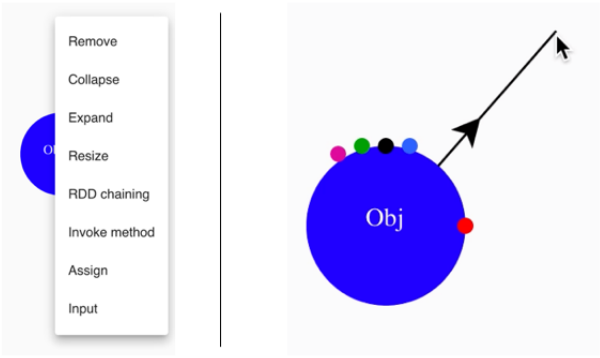
V prvom spôsobe je užívateľovi po kliknutí na zvolený uzol poskytnuté kontextové menu, ktoré okrem iných metód manipulácie s uzlom ponúka možnosť modifikácie veľkosti daného uzlu alebo vytvorenie hrán. Graf obsahuje štyri typy hrán, preto kontextové menu obsahuje štyri možnosti vytvorenia hrany. Výhodou kontextového menu je možnosť textového popisu položiek menu, respektíve užívateľských akcií. Na druhej strane samotné vytvorenie hrany predchádza sled udalostí nutný vykonať, čo sa pri vytvorení väčšieho počtu hrán môže javiť ako negatívny aspekt.

V druhej metóde sa po kliknutí na vybraný uzol grafu zobrazia kruhy po obvode vybraného uzlu. Štyri z nich indikujú vytvorenie hrán, ktoré sa udeje ako náhle užívateľ na jeden z nich klikne a následne ťahaním kurzoru vytvorí hranu. Piaty slúži na zmenenie veľkosti vybrané uzlu ťahaním kurzoru. Výhodou uvedenej metódy je jednoduchý spôsob vyvolania akcie užívateľa bez nutnosti využitia kontextového menu. Bez kontextového menu však odpadá textový popis akcií, čo vnáša nejednoznačnosť funkcionality prvkov GUI. Taktiež vizuálne prevedenie nie je úplne atraktívne. Oba spôsoby prevedenia sú zobrazené na obrázku 7.2.

Medzi oslovenými testovacími užívateľmi prevažovala voľba druhého riešenia. Napriek jeho uvedeným nevýhodám ocenili jednoduchosť vytvorenia hrany. Po spracovaní podnetov od užívateľov bola pridaná do GUI legenda vysvetlujúca funkcionalitu jednotlivých kruhov.

7.2 Testovanie generovania kódu

Podstatná časť testovania, nie len generovania, ale celej aplikácie prebiehala počas implementácie. Po dokončení implementácie generovania výsledného kódu bola aplikácia testo-



Obrázek 7.2: Vytvorenie hrany pomocou kontextového menu vľavo. Vytvorenie hrany druhým spôsobom vpravo.

vaná na viacerých príkladoch. Príklady boli vytvárané na základe zdrojového kódu v jazyku Scala existujúcej Apache Spark aplikácie. Podľa tohto kódu bol vytvorený graf, a z grafu vygenerovaný zdrojový kód. Následne bol spustený vzorový kód, vygenerovaný kód a ich výstup porovnaný. Pri jednotlivých príkladoch sú uvedené vizuálne ukážky oboch typov kódov, výstupy aplikácie po ich spustení a ukážka grafu, z ktorej bol kód generovaný. V poslednom teste je vytvorená Apache Spark aplikácia bez vzorových riešení.

Pri testovaní boli použité nasledovné verzie jazyku Scala a jeho knižníc:

- jazyk Scala: *2.11.8*
- knižnica `spark-core`: *2.4.0*
- knižnica `spark-sql`: *2.4.0*
- knižnica `spark-mllib`: *2.4.1*
- knižnica `spark-streaming`: *2.4.1*
- knižnica `spark-graphx`: *2.4.1*
- `sbt`: *0.13.18*

Kvôli prehľadnosti výstupov aplikácie boli vypnuté implicitné záznamy, ktoré Apache Spark vypisuje na štandardný výstup. Pri vytváraní `SparkSession`[1] bola využitá konfigurácia `master("local")`, ktorú vzorové príklady neobsahovali. Ukážky vzorových zdrojových kódov taktiež neobsahujú komentáre. Okrem vyššie uvedených zmien neboli vzorové príklady inak modifikované.

Test číslo 1

V prvom testovacom príklade bol vzorový zdrojový kód Apache Spark aplikácia, ktorá vypočíta približnú hodnotu π. Na základe vzorového kódu bol vymodelovaný graf A.2. Graf je v porovnaní so zdrojovým kódom pomerne zložitý. Problém zložitosti grafu bude adresovaný v teste číslo 2.

[1] Vstupný bod, ktorý je potrebné vytvoriť v každej Apache Spark aplikácií.

```scala
import scala.math.random
import org.apache.spark.sql.SparkSession

/** Computes an approximation to pi */
object SparkPi {
  def main(args: Array[String]) {
    val spark = SparkSession
      .builder
      .appName("Spark Pi")
      .master("local")
      .getOrCreate()
    val slices = if (args.length > 0) args(0).toInt else 2
    val n = math.min(100000L * slices, Int.MaxValue).toInt // avoid overflow
    val count = spark.sparkContext.parallelize(1 until n, slices).map { i =>
      val x = random * 2 - 1
      val y = random * 2 - 1
      if (x*x + y*y <= 1) 1 else 0
    }.reduce(_ + _)
    println(s"Pi is roughly ${4.0 * count / (n - 1)}")
    spark.stop()
  }
}
```

Výpis 7.1: Výpis vzorového príkladu, ktorý spočíta približnú hodnotu π a vypíše ju na štandardný výstup[3].

Príloha B.1 zobrazuje vygenerovaný zdrojový kód z grafu A.2, ktorý modeluje vzorový kód uvedený v ukážke 7.1. Ako je možné vidieť vzorový a vygenerovaný kód sa líšia len mierne. Rozdiel v poradí prvých dvoch riadkov vyplýva z poradia, akým užívateľ pridal konkrétne triedy do textového pola slúžiaceho na pridávanie tried do aplikácie. Ďalší rozdiel je v odsadení zdrojového kódu. Generovaný kód využíva na odsadenie príkazov biely znak`\t` namiesto dvoch medzier. Reťazenie RDD operácií objektu `SparkSession` je zobrazené na jednom riadku oproti vzorovému riešeniu, ktoré v niektorých prípadoch uvádza každú operáciu na novom riadku. Posledným rozdielom je, že vygenerovaný kód pri definovaní premenných využíva kľúčové slovo `var` namiesto `val`, použitého vo vzorovom riešení. Žiaden zo zmienených rozdielov však nevplýva na funkcionalitu výslednej aplikácie. Výstup programov reprezentovaných vzorovým a generovaným kódom je totožný. Ukážka 7.2 zobrazuje štandardný výstup oboch programov.

```
Pi is roughly 3.1414957074785375
Process finished with exit code 0
```

Výpis 7.2: Výstup prvej testovanej aplikácie.

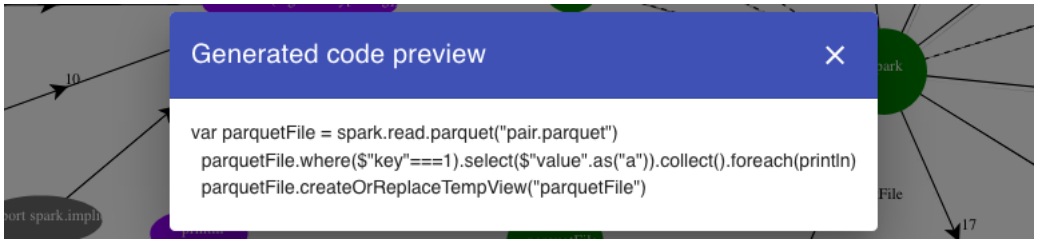
Test číslo 2

Druhý test sa zameriava na modelovanie väčšieho počtu RDD operácií ako v teste číslo 1. Jedná sa o Apache Spark aplikáciu, ktorá definuje RDD schému pomocou triedy `Record`. Schéma, respektíve trieda, obsahuje stĺpce `key` a `value`. Po vytvorení schémy je na ňu aplikovaný SQL dotaz `SELECT` a agregačná funkcia `COUNT`. Následne sú na SQL dotazy

[3]Príklad bol prevzatý z github repozitára spark/exmaples.

aplikované niektoré RDD operácie. V poslednom rade sú RDD zapísané do *Parquet file*[4] a nad ním sú opäť vykonané modifikácie. Testovanie taktiež zahŕňa spôsob zjednodušenia grafu, ktorý sa s narastajúcim počtom uzlov stáva neprehľadným.

Na základe vzorového kódu B.2 bol vymodelovaný graf A.3, ktorý zobrazuje využitie schovávania uzlov grafu. Aktuálne zobrazená je vždy len prvá RDD operácia invokovaná premennou alebo objektom. Ostatné RDD operácie v zreťazení boli schované. Celý graf bez využitia schovávania uzlov je pribalený v archíve na CD. Následne bola otestovaná funkcia zobrazenia vygenerovaného kódu pre jednotlivé uzly grafu. V prípade, že užívateľ klikne na uzol a zvolí funkciu*preview* z kontextového menu, vytvorí sa ukážka vygenerovaného zdrojového kódu, ktorý obsahuje užívateľom kliknutý uzol. Ukážka počíta aj so schovanými uzlami, čím umožňuje lepšiu orientáciu v grafe. Táto funkcionalita je predvedená na obrázku 7.3.



Obrázek 7.3: Ukážka časti vygenerovaného kódu uzlu reprezentujúceho premennú *parquetFile*.

V grafe bolo nutné explicitne uviesť, že premenná `spark` je konštanta pridaním kľúčového slova `val`, z dôvodu využitia implicitných konverzií `import spark.implicits._`. V opačnom prípade výsledný zdrojový kód nie je možné skompilovať. Rozdiely vo vzorovom a vygenerovanom kóde zostávajú rovnaké ako v teste číslo 1. Konkrétne sú to poradie pridania tried v prvých riadkoch kódu a odsadenie jednotlivých častí programu. Pribudol rozdiel vytvorenia triedy `case class Record`, ktorý sa vo vzorovom kóde nachádza na začiatku súboru. Vo vygenerovanom kóde je definícia triedy na konci súboru a naviac obsahuje zátvorky identifikujúce prázdne telo triedy. Rovnako ako v prvom teste rozdiely v zdrojových kódoch neovplyvňujú funkcionalitu programu, ktorý reprezentujú. Výstup programu v oboch prípadoch je totožný a znázornený výpisom 7.3.

[4]Parquet file je v Apache Spark stĺpcový formát súboru poskytujúci optimalizácie dotazov, ktoré sú nad ním prevádzané.

```
Result of SELECT *:
[1,val_1]
...
[100,val_100]
COUNT(*): 100
Result of RDD.map:
Key: 1, Value: val_1
...
Key: 9, Value: val_9
Parquet file:
[1,val_1]
...
[100,val_100]
Process finished with exit code 0
```

Výpis 7.3: Skrátený výstup druhej testovanej aplikácie. Tri bodky reprezentujú vynechanú časť výstupu. Prvá časť výstupu zobrazuje SQL dotaz `SELECT` nasledovaný agregačnou funkciou `COUNT`. Po agregačnej funkcií nasleduje výpis RDD operácií a opäť SQL dotazy aplikované na *Parquet file*.

Test číslo 3

Tretí test je zameraný na spracovanie Big dat a na rozdiel od prechádzajúcich príkladov nebol využitý vzorový zdrojový kód. RDD transformácie boli aplikované na súbor vo formáte CSV obsahujúci týždenné štatistiky zberu obilia kanadskej vlády[5]. Súbor má veľkosť 13.8 MB, obsahuje 65 535 riadkov a 10 stĺpcov.

Prvý krok testovania spočíval v načítaní súboru ako DataFrame a vypísaní schémy pre lepšiu orientáciu s dátami. Schéma je zobrazená vo výpise 7.4.

```
root
 |-- grain_week: string (nullable = true)
 |-- crop_year: string (nullable = true)
 |-- week_ending_date: string (nullable = true)
 |-- worksheet: string (nullable = true)
 |-- metric: string (nullable = true)
 |-- period: string (nullable = true)
 |-- grain: string (nullable = true)
 |-- grade: string (nullable = true)
 |-- region: string (nullable = true)
 |-- Ktonnes: string (nullable = true)
```

Výpis 7.4: Schéma vstupného súboru.

Po zobrazení schémy boli vybrané stĺpce *region* a *grain*. Výstup výberu je zobrazený v prílohe B.5. Príloha B.6 zobrazuje aplikovanie RDD operácie filter na nový výber. V poslednom rade bola aplikovaná agregačná funkcia *count* spolu s filtrom, ktorých výsledok je zobrazený vo výpise 7.5. Vyššie popísaná aplikácia bola vygenerovaná z modelu zobrazeného v prílohe A.4. Príloha A.4 zobrazuje ten istý graf s využitím operácie *expand*,

[5]Vstupný súbor je voľne dostupný na webových stránkach kanadskej vlády.

ponúkanú modelovacou aplikáciou. Operácia bola použitá na uzol *df.select*. Jej výsledkom je zobrazenie všetkých uzlov s spojených s daným uzlom.

Testovacie príklady pokrývajú len malú časť Apache Spark aplikácií. Vybrané boli príklady využívané zreťazenie RDD operácií, na ktoré sa výsledná aplikácia a model pomocou nej vytvorený zameriava. Programy reprezentované vygenerovanými kódmi z grafu zachovávajú rovnakú funkcionalitu ako programy vytvorené zo vzorových kódov. Všetky vzorové a vygenerované zdrojové kódy, grafy a výstupy použité pri testovaní sú súčasťou archívu na CD pribaleného s textom diplomovej práce.

```
765
Process finished with exit code 0
```

Výpis 7.5: Výstup zobrazuje počet kiloton vypestovanej pšenice v regióne Alberta za rok 2017-2018. Výsledok bol dosiahnutý aplikovaním RDD operácií `select` na schému 7.4 získanú zo vstupného súboru. Následne pomocou operácií `filter` boli vybrané iba potrebné informácie. V poslednom rade bola aplikovaná agregačná operácia `count`, ktorá spočítala dosiahnutú hodnotu.

Kapitola 8

Záver

Cieľom diplomovej práce bolo zoznámiť sa s prostredím Apache Spark pre distribuované spracovanie Big data a s programovacími jazykmi, v ktorých sa dá framework použiť. Ďalšou fázou bolo zoznámenie sa s prístupom modelom riadeného vývoja (Model Driven Development, MDD).Následne preskúmať možnosti a existujúce projekty pre modelovanie úloh spracovania dát zamerané na Big data. Po preskúmaní možností a oboznámení s danou problematikou navrhnúť alebo upraviť existujúci grafický modelovací jazyk pre úlohy spracovania Big data v prostredí frameworku Apache Spark. Modelovací jazyk musí umožňovať generovanie zdrojového kódu Spark aplikácií. V prvej časti textu práce je predstavený framework Apache Spark, ktorý adresuje nedostatky služby Apache Hadoop. Čitateľ je oboznámený so základnými typmi abstrakcie v prostredí Apache Spark. Počínajúc od RDD používaných hlavne v starších verziách až po Dataset, ktorý doplňuje nedostatky RDD a využíva sa v novších verziách frameworku. Popísaná je štruktúra a predstavené sú základné knižnice používané pri vytváraní aplikácií v prostredí Spark. Všetky vyššie zmienené informácie spolu s programovacími jazykmi, v ktorých je možné framework využívať sú obsiahnuté v kapitole 2.

Kapitola 3 približuje paradigma modelom riadeného vývoja a modelom riadenej architektúry spolu s výhodami a nevýhodami, ktoré z nich vyplývajú. Predstavené boli tri existujúce riešenia založené na modelom riadenom vývoji. Prvým z nich je *Executable UML* založené na matematickom modely modifikujúce štandardný jednotný modelovací jazyk UML. Executable UML (xUML) umožňuje pomocou aplikácií tretích strán generovať zdrojový kód niektorých programovacích jazykov. Zvyšné dve sú cloudové služby od spoločnosti Google zamerané na spracovanie Big data. *Dataflow* ponúka spôsob ako manipulovať so vstupnými dátami aplikovaním rôznych transformácií podobne ako Apache Spark využívajúci RDD. Výsledný program sa skladá so vzájomne interagujúcich častí, ktoré je síce nutné naprogramovať, ale užívateľovi je poskytnutá aj jeho grafická reprezentácia. Posledným existujúcim riešením je služba *Dataprep* poskytujúca možnosť vizualizácie a manipulácie so štruktúrovanými a neštruktúrovanými dátami za účelom uľahčenia dátovej analýzy. Na rozdiel od prvých dvoch riešení Dataprep nevytvára model, ale poskytuje grafické užívateľské rozhranie na vizualizáciu dát a ich modifikáciu.

Po teoretickom úvode nasleduje kapitola 5 popisujúca vlastný návrhu grafického modelovacie jazyka. Rozhodnutie vytvoriť vlastný model a nevyužiť dostupné modelovacie platformy vyplýva z faktu, že neboli nájdené vhodné prostriedky pre tento špecifický problém. Ďalším dôvodom je poskytnúť užívateľovi interaktívne modelovacie prostredie, ktoré nie je závislé na verzií Apache Spark. V závere kapitole sú navrhnuté technológie, ktoré by bolo vhodné použiť pri implementácií aplikácie.

V kapitole 6 sú technológie z návrhu popísané podrobnejšie. Text sekcie 6.1 obsahuje najdôležitejšie časti implementácie a dizajnu grafického užívateľského rozhrania. Následne je uvedený spôsob integrácie dvoch hlavných knižníc *React* a *Data driven documents(D3)*, využitých pri implementácií aplikácie. Po priblížení spôsobu vytvorenia grafického rozhrania sa text zameriava na komunikáciu medzi klientom a serverom. Na konci kapitoly je popísaná implementácia generovania zdrojového kódu z modelu a rozdiely v návrhu a výslednej aplikácií.

Funkčnosť výslednej aplikácie bola v prvom rade otestovaná vytvorením modelov podľa vzorových zdrojových kódov. Vzorové a vygenerované kódy boli vizuálne a funkčne porovnané. Následne bolo uskutočnené testovanie bez vzorových príkladov. Výsledkom bola aplikácia využívajúca rôzne selekcie, filtre a agregačnú funkciu na voľne dostupný zdroj Big data. Okrem testovania funkcionality aplikácie prebiehalo aj testovanie grafického užívateľského rozhrania a užívateľskej skúsenosti. Podstatná časť sa odohrávala na predmete *UXIa*. Testovanie je uvedené v kapitole 7.

Zhrnutie výsledku a výhľad do budúcna

Vytvorená aplikácia bola podrobne otestovaná a zverejnená na serveri GitHub[1]. Výsledný program sa skladá z prostredia pre tvorbu grafu a serveru doplňujúceho funkcionalitu modelovacieho nástroja. Každopádne prostredie pre tvorbu grafu je funkčné aj bez serverovej časti, ale s obmedzenou funkcionalitou. K požiadavkám zo zadania na výslednú aplikáciu, bola pridaná možnosť zobraziť si výsledný kód počas procesu modelovania. Ďalšou pridanou funkciu aplikácie je zoznam dostupných metód jednotlivých modelovaných tried a objektov. Avšak implementácia tejto funkcionality nebola dokončená.

Graf vytvorený v aplikácií je pre komplexnejšie Apache Spark programy zložitý. Tento problém sa snažia adresovať dve funkcie aplikácie. Prvou z nich je možnosť schovávať časti grafu a druhá funkcia umožňuje užívateľovi zobraziť generovaný kód častí grafu.

Do budúcna by bolo vhodné rozšíriť funkcionalitu o možnosť uložiť graf do databázi, prípadne model exportovať alebo importovať. Ideálnym rozšírením funkcionality by bola schopnosť vedieť graf vymodelovať priamo zo zdrojových kódov jednotlivých Apache Spark programov, keďže vytvorený model je skôr vhodný na zobrazenie závislosti v zdrojovom kóde ako na priame modelovanie aplikácií.

[1] Repozitár aplikácie — https://github.com/butoramatus/dip

Literatura

[1] *Akka HTTP*. [Online; navštíveno 28.12.2018].
URL https://doc.akka.io/docs/akka-http/current/

[2] *Apache Spark: 3 Reasons Why You Should Not Use RDDs* . [Online; navštíveno 5.1.2018].
URL https://dzone.com/articles/apache-spark-3-reasons-why-you-should-not-use-rdds

[3] *A Tale of Three Apache Spark APIs: RDDs vs DataFrames and Datasets*. [Online; navštíveno 5.1.2018].
URL https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

[4] *Cloud Dataflow Documentation*. [Online; navštíveno 13.3.2018].
URL https://cloud.google.com/dataflow/docs/

[5] *Cloud Dataprep by Trifacta*. [Online; navštíveno 15.3.2018].
URL https://cloud.google.com/dataprep/

[6] *D3.js*. [Online; navštíveno 4.1.2019].
URL https://d3js.org/

[7] *Executable UML (xUML)*. [Online; navštíveno 25.4.2018].
URL https://executableuml.org/

[8] *MDA Specifications*. [Online; navštíveno 3.1.2018].
URL https://www.omg.org/mda/specs.htm

[9] *Modelování REST APIs*. [Online; navštíveno 4.1.2018].
URL https://www.zdrojak.cz/clanky/modelovani-rest-apis/

[10] *RDD Programming Guide*. [Online; navštíveno 23.11.2018].
URL https://spark.apache.org/docs/latest/rdd-programming-guide.html

[11] *React*. [Online; navštíveno 25.4.2018].
URL https://reactjs.org/

[12] *Run a Big Data Text Processing Pipeline in Cloud Dataflow*. [Online; navštíveno 11.3.2018].
URL
https://codelabs.developers.google.com/codelabs/cloud-dataflow-starter/

[13] *Spark SQL, DataFrames and Datasets Guide*. [Online; navštíveno 24.11.2018].
URL https://spark.apache.org/docs/latest/sql-programming-guide.html

[14] *Spark Streaming Programming Guide*. [Online; navštíveno 25.11.2018].
URL
https://spark.apache.org/docs/latest/streaming-programming-guide.html

[15] *What is Material Design?* [Online; navštíveno 29.4.2018].
URL
https://www.interaction-design.org/literature/topics/material-design

[16] *What is Model Driven Development (MDD)*. [Online; navštíveno 3.1.2018].
URL https://www.mendix.com/model-driven-development/

[17] Benjamin Bengfor, J. K.: *In-memory Computing with Spark*. [Online; navštíveno 22.11.2018].
URL https://www.oreilly.com/library/view/data-analytics-with/9781491913734/ch04.html

[18] Béder, M.: *Spracování síťové komunikace v prostředí Apache Spark*. Diplomová práce, Vysoké učení technické, Fakulta informačních technológií, Česká republika, Brno, 2017, https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=159661.

[19] Bělehrádek, S.: *Modelem řízený vývoj android aplikací*. Diplomová práce, Vysoké učení technické, Fakulta informačních technológií, Česká republika, Brno, 2017, https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=159661.

[20] Chambers, B.; Zaharia, M.: *Spark: The Definitive Guide: Big Data Processing Made Simple*. O'Reilly Media, 2018, ISBN 9781491912300.

[21] Frampton, M.: *Mastering Apache Spark*. Packt Publishing, 2015, ISBN 978-1-78398-714-6.

[22] Iglesias, M.: *Bringing Together React, D3, And Their Ecosystem*. [Online; navštíveno 20.3.2019].
URL https://www.smashingmagazine.com/2018/02/react-d3-ecosystem/

[23] Karau, H.; Konwinski, A.; Wendell, P.; aj.: *Learning Spark*. O'Reilly Media, Inc, 2015, ISBN 9781449358624.

[24] Laskowski, J.: *Mastering Apache Spark 2.3.2*. [Online; navštíveno 22.11.2018].
URL https://legacy.gitbook.com/book/jaceklaskowski/mastering-apache-spark/details

[25] Leon Starr, S. M., Andrew Mangogna: *Models to Code: With No Mysterious Gaps*. Apress, 2017, ISBN 9781484222171.

[26] Marco Brambilla, M. W., Jordi Cabot: *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2017, ISBN 9781627059886.

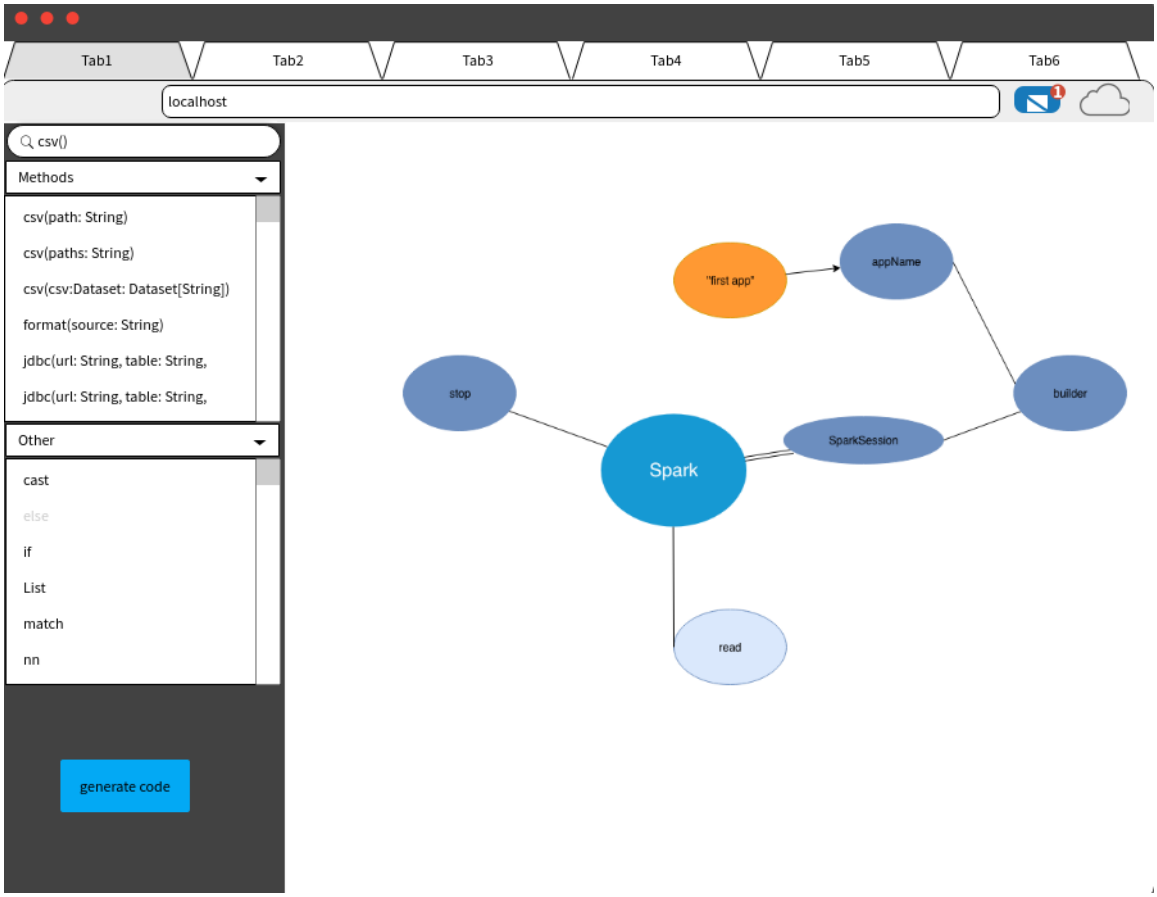
[27] Rouse, M.: *XMI (XML Metadata Interchange)* . [Online; navštíveno 3.1.2018].
URL https://searchmicroservices.techtarget.com/definition/XMI-XML-Metadata-Interchange

[28] Ruben Picek, V. S.: *Model Driven Development – Future or Failure of Software Development?* . 2008.

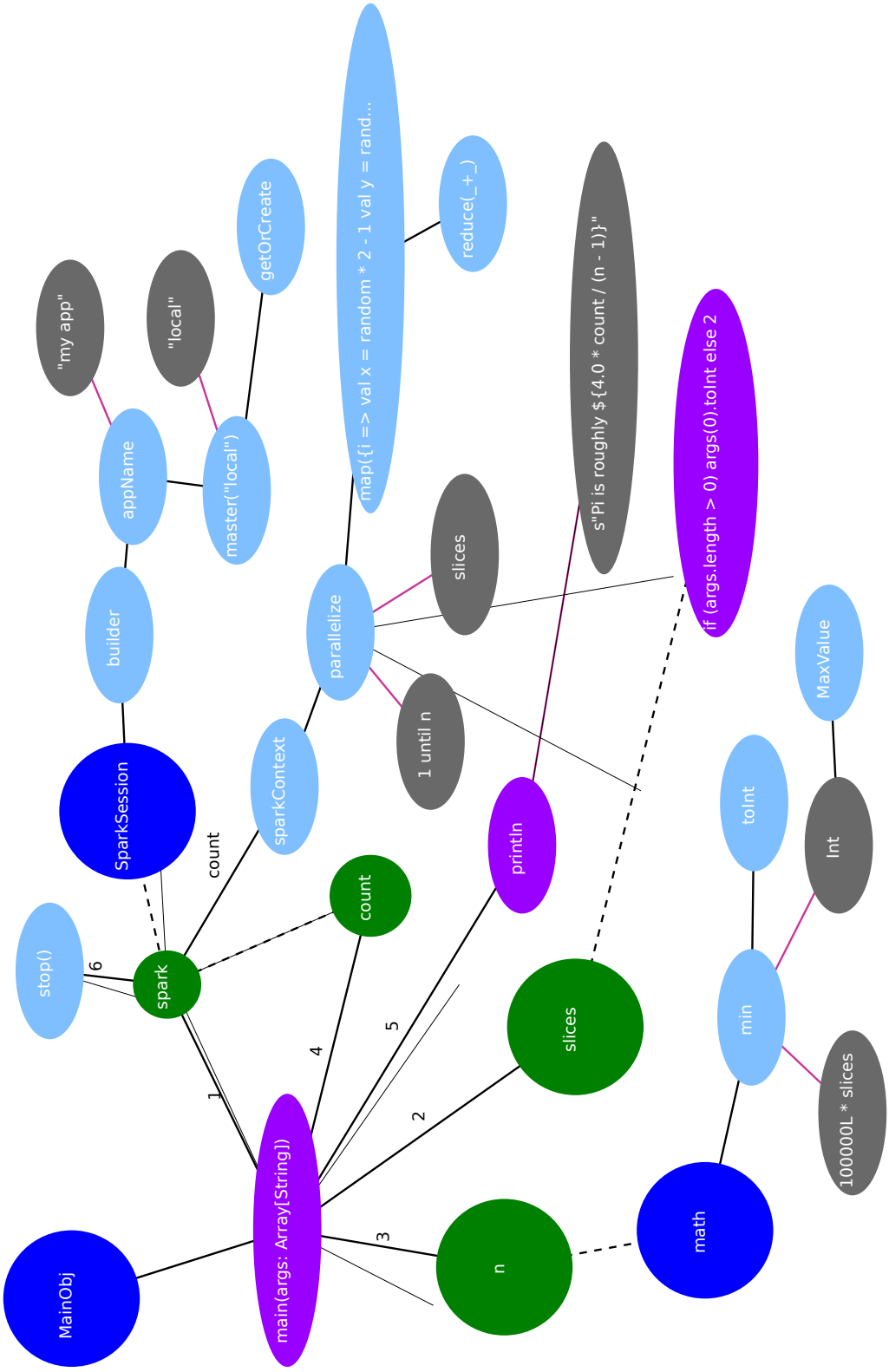
[29] Sivashanmugam, K.: *Developing Apache Spark Applications in .NET using Mobius.* [Online; navštíveno 22.11.2018].
URL https://databricks.com/blog/2016/08/03/developing-apache-spark-applications-in-net-using-mobius.html

Příloha A

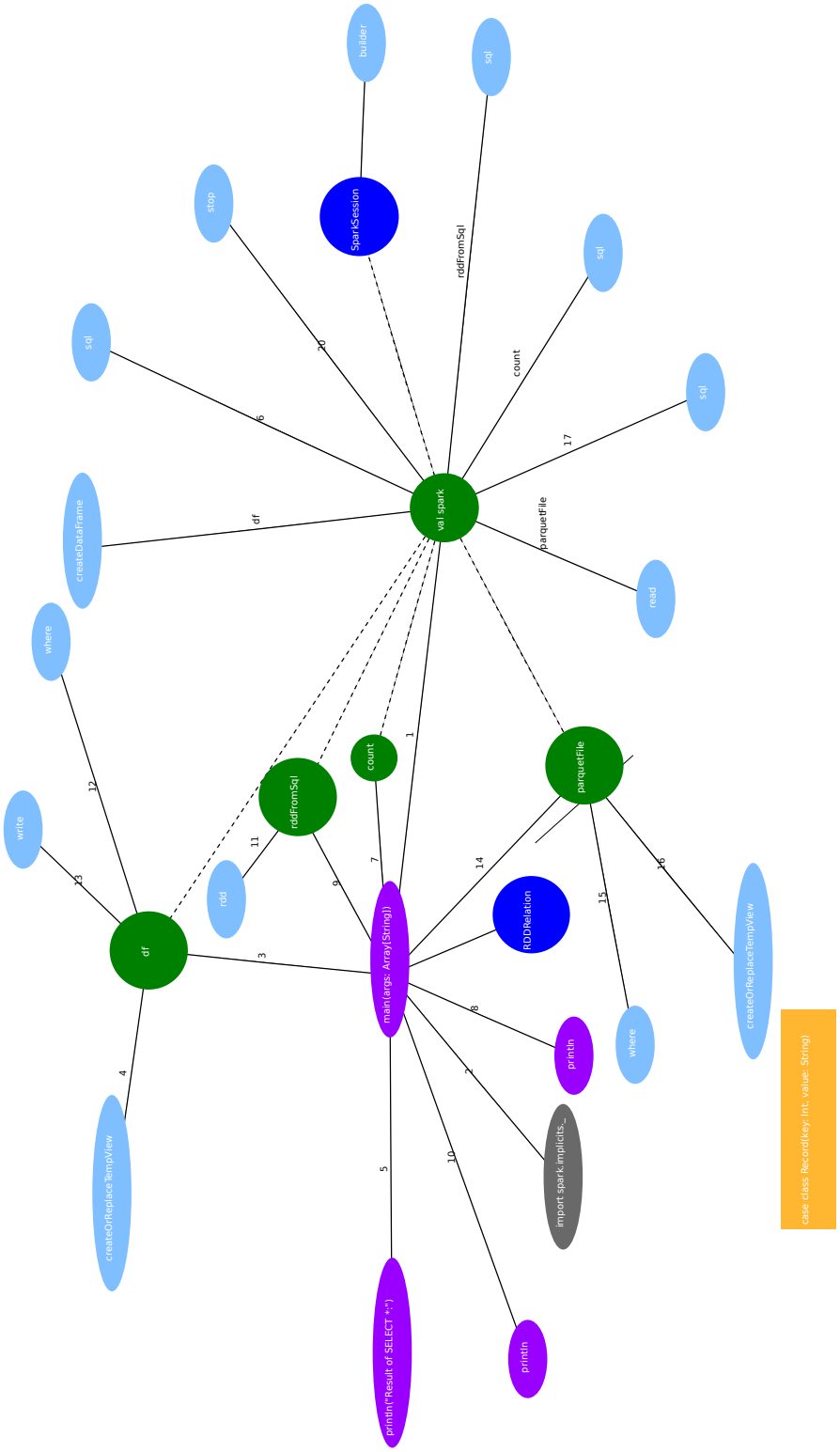
Obrázky



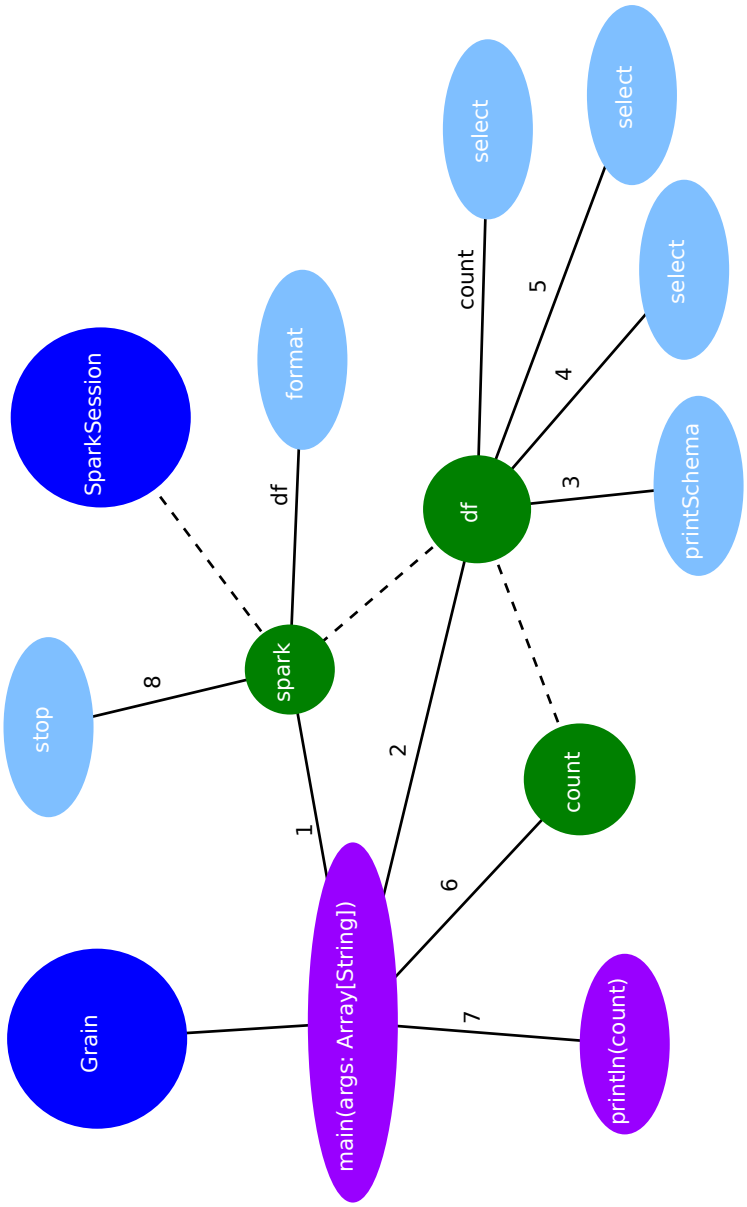
Obrázek A.1: Mockup grafického užívateľského rozhrania.



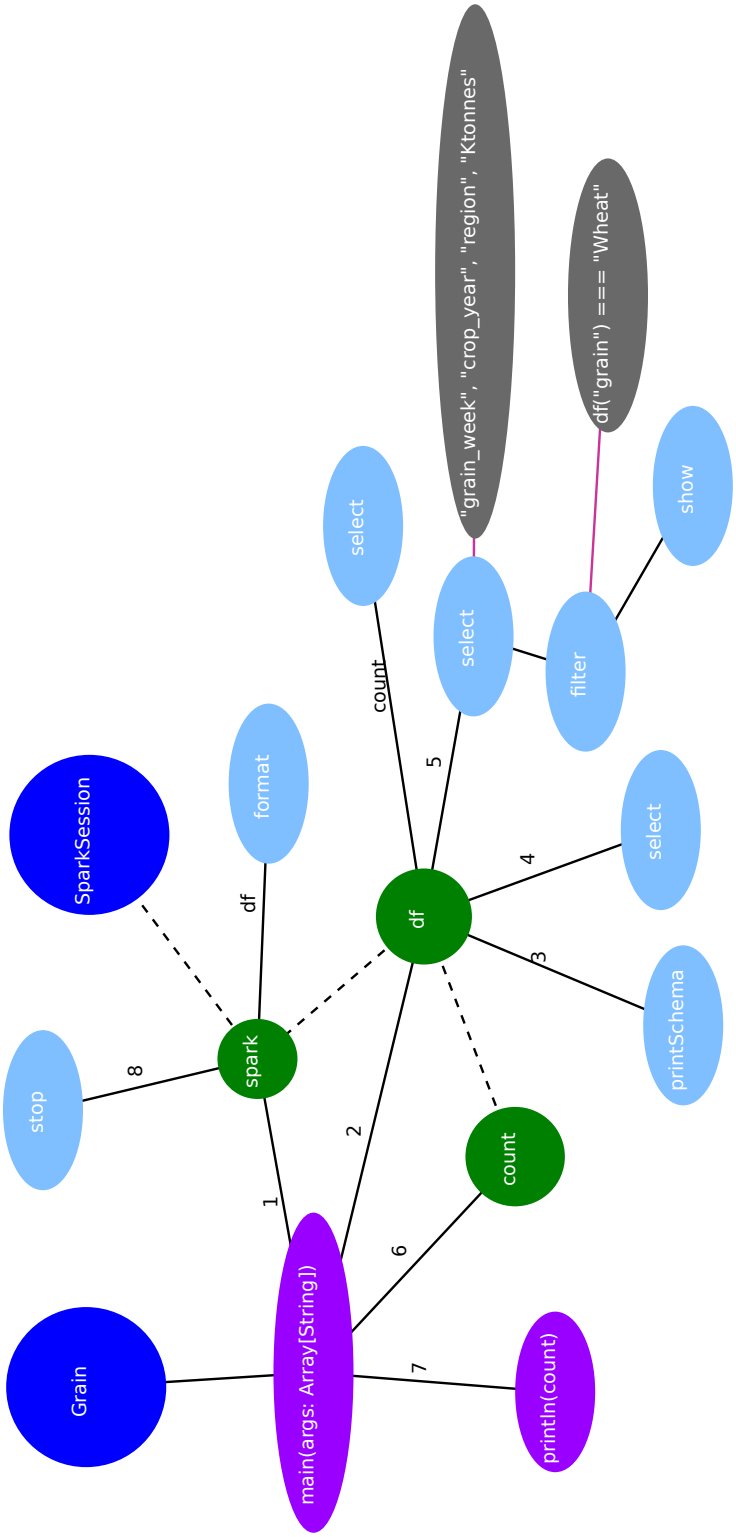
Obrázek A.2: Graf modelujúci Apache Spark aplikáciu reprezentovanú zdrojovým kódom 7.1.



Obrázek A.3: Graf modelujúci Apache Spark aplikáciu reprezentovanú zdrojovým kódom B.2 s využitím schovávania uzlov.



Obrázek A.4: Graf modelujúci Apache Spark aplikáciu popísanú v teste číslo 3 7.2. Graf využíva možnosť aplikácie schovávať uzly.



Obrázek A.5: Graf modelujúci Apache Spark aplikáciu popísanú v teste číslo 3 7.2. Graf využíva možnosť aplikácie schovávať uzly. Na uzol `df.select` bola aplikovaná funkcia *expand*, poskytovaná aplikáciou. Zobrazujú sa tak všetky uzly spojené z uzlom `df.select`.

Příloha B

Výpisy

```
import org.apache.spark.sql.SparkSession
import scala.math.random

object SparkPi {
  def main(args: Array[String]) {
    val spark = SparkSession.builder.appName("Spark Pi").master("local")
      .getOrCreate()
    val slices = if (args.length > 0) args(0).toInt else 2
    val n = math.min Int.MaxValue, 100000L * slices).toInt
    val count = spark.sparkContext.parallelize(1 until n, slices)
      .map {
         i =>val x = random * 2 - 1
         val y = random * 2 - 1
         if (x*x + y*y <= 1) 1 else 0
      }.reduce(_ + _)
    println(s"Pi is roughly ${4.0 * count / (n - 1)}")
    spark.stop()
  }
}
```

Výpis B.1: Test 1: Vygenerovaný kód z grafu A.2.

```
import org.apache.spark.sql.SaveMode
import org.apache.spark.sql.SparkSession

case class Record(key: Int, value: String)

object RDDRelation {
  def main(args: Array[String]) {
    val spark = SparkSession
      .builder
      .appName("Spark Examples")
      .config("spark.some.config.option", "some-value")
      .getOrCreate()

    import spark.implicits._

    val df = spark.createDataFrame((1 to 100).map(i => Record(i, s"val_$i")))
    df.createOrReplaceTempView("records")

    println("Result of SELECT *:")
    spark.sql("SELECT * FROM records").collect().foreach(println)

    val count = spark.sql("SELECT COUNT(*) FROM
        records").collect().head.getLong(0)
    println(s"COUNT(*): $count")

    val rddFromSql = spark.sql("SELECT key, value FROM records WHERE key < 10")

    println("Result of RDD.map:")
    rddFromSql.rdd.map(row => s"Key: ${row(0)}, Value: ${row(1)}")
      .collect().foreach(println)

    df.where($"key" ===
        1).orderBy($"value".asc).select($"key").collect().foreach(println)

    df.write.mode(SaveMode.Overwrite).parquet("pair.parquet")

    val parquetFile = spark.read.parquet("pair.parquet")

    parquetFile.where($"key" === 1).select($"value".as("a"))
      .collect().foreach(println)

    parquetFile.createOrReplaceTempView("parquetFile")
    spark.sql("SELECT * FROM parquetFile").collect().foreach(println)

    spark.stop()
  }
}
```

Výpis B.2: Test 2: Vzorový zdrojový kód druhého testu.

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.SaveMode

object RDDRelation {
   def main(args: Array[String]) {
      val spark = SparkSession.builder.appName("my app").master("local")
         .getOrCreate
      import spark.implicits._
      var df = spark.createDataFrame((1 to 100).map((i => Record(i, s"val_$i"))))
      df.createOrReplaceTempView("records")
      println("Result of SELECT *:")
      spark.sql("SELECT * FROM records").collect.foreach(println)
      var count = spark.sql("SELECT COUNT(*) FROM records")
         .collect().head.getLong(0)
      println(s"COUNT(*): $count")
      var rddFromSql = spark.sql("SELECT key, value FROM records WHERE key < 10")
      println("Result of RDD.map:")
      rddFromSql.rdd.map(row =>s"Key:${row(0)},Value:${row(1)}")
         .collect().head.getLong(0)
      df.where($"key" ===
          1).orderBy($"value".asc).select($"key").collect().foreach(println)
      df.write.mode(SaveMode.Overwrite).parquet("par.parquet")
      var parquetFile = spark.read.parquet("pair.parquet")
      parquetFile.where($"key"===1).select($"value".as("a")).collect().foreach(println)
      parquetFile.createOrReplaceTempView("parquetFile")
      spark.sql("SELECT * FROM parquetFile").collect().foreach(println)
      spark.stop
   }
}

case class class Record(key: Int, value: String) {
}
```

Výpis B.3: Test 2: Vygenerovaný kód z grafu A.3.

```
import org.apache.spark.sql.SparkSession

object Grain {
   def main(args: Array[String]) {
      val spark = SparkSession.builder.appName("big data test").master("local")
            .getOrCreate()
      val df = spark.read.format("csv").option("header",
          "true").load("src/resources/test.csv")
        df.printSchema()
        df.select("region", "grain").show()
        df.select("grain_week", "crop_year", "region", "Ktonnes")
         .filter(df("grain") === "Wheat").show()
        val count = df.select("Ktonnes").filter(df("grain") === "Wheat")
         .filter(df("region") === "Alberta").count()
        print(count)
        spark.stop()
    }
}
```

Výpis B.4: Test 3: Vygenerovaný kód z grafu A.4, popísanom v teste číslo 3 7.2.

```
+----------------+-----------+
|          region|      grain|
+----------------+-----------+
|        Manitoba|      Wheat|
|    Saskatchewan|      Wheat|
|         Alberta|      Wheat|
|British Columbia|      Wheat|
|        Manitoba|     Barley|
|    Saskatchewan|     Barley|
|         Alberta|     Barley|
|British Columbia|     Barley|
|        Manitoba|Amber Durum|
|    Saskatchewan|Amber Durum|
|         Alberta|Amber Durum|
|British Columbia|Amber Durum|
|        Manitoba|    Lentils|
|    Saskatchewan|    Lentils|
|         Alberta|    Lentils|
|British Columbia|    Lentils|
|        Manitoba| Chick Peas|
|    Saskatchewan| Chick Peas|
|         Alberta| Chick Peas|
|British Columbia| Chick Peas|
+----------------+-----------+
only showing top 20 rows
```

Výpis B.5: Výstup po aplikovaní operácie `select("region", "grain")` na DataFrame v sekcií 7.2.

```
+----------+---------+-----------------+-------+
|grain_week|crop_year|           region|Ktonnes|
+----------+---------+-----------------+-------+
|         1|2017-2018|         Manitoba|    1.4|
|         1|2017-2018|     Saskatchewan|   19.6|
|         1|2017-2018|          Alberta|    5.9|
|         1|2017-2018| British Columbia|    0.5|
|         1|2017-2018|         Manitoba|    1.2|
|         1|2017-2018|     Saskatchewan|    8.4|
|         1|2017-2018|          Alberta|    1.5|
|         1|2017-2018| British Columbia|    1.0|
|         1|2017-2018|         Manitoba|    1.4|
|         1|2017-2018|     Saskatchewan|   19.5|
|         1|2017-2018|          Alberta|    5.9|
|         1|2017-2018| British Columbia|    0.5|
|         1|2017-2018|         Manitoba|    1.2|
|         1|2017-2018|     Saskatchewan|    8.4|
|         1|2017-2018|          Alberta|    1.5|
|         1|2017-2018| British Columbia|    1.0|
|         1|2017-2018|Canadian Domestic|    4.6|
|         1|2017-2018|Process Elevators|      0|
|         1|2017-2018|          Pacific|    5.5|
|         1|2017-2018|        Churchill|      0|
+----------+---------+-----------------+-------+
only showing top 20 rows
```

Výpis B.6: Výstup vznikol aplikovaním operácie `select("grain_week", "crop_year", "region", "Ktonnes").filter(df("grain") === "Wheat")` na DataFrame v sekcií 7.2.

Příloha C

Obsah CD

Kapitola popisuje obsah CD, ktoré je priložené k originálnej verzií diplomovej práce. Priložené CD obsahuje diplomovú prácu vo formáte PDF, zdrojové súbory LaTeX a zdrojové kódy výslednej aplikácie. Taktiež obsahuje dokumentáciu k vytvorenej aplikácií a súbor *README.md*, v ktorom je obsiahnutý postup inštalácie aplikácie. Archív má nasledovnú štruktúru:

- *document* — adresár so správou vo formáte PDF a zdrojové kódy LaTeX,
- *src* — adresár so zdrojovými kódmi aplikácie, súbor *README.md* a dokumentáciu aplikácie,