UNIVERZITA PARDUBICE

Fakulta elektrotechniky a informatiky

DIPLOMOVÁ PRÁCE

2020 Bc. Martin Bárta

Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Datamining výsledků experimentů

Diplomová práce

2020 Bc. Martin Bárta

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2019/2020

ZADÁNÍ DIPLOMOVÉ PRÁCE
(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Martin Bárta**
Osobní číslo:	**I17201**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Téma práce:	**Datamining výsledků experimentů**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování

Numerické experimenty s algoritmy, jako jsou např. algoritmy genetického programování, vedou k produkci rozsáhlých souborů dat, které je obtížné manuálně analyzovat. Přitom jsou ovlivněny mnoha vlivy, jako je např. zvolený generátor pseudonáhodných čísel a jejich rozdělení, soubor funkcí (stavebních bloků), algoritmus, ale i vlivy operačního systému. Jednou z možností jejich zpracování je použití vhodného open source data-miningového nástroje, který by mohl pomoci odhalit skryté souvislosti. Cílem práce je nalézt takovýto nástroj a ověřit jeho vhodnost. V teoretické části se očekává analyzujte dostupného souboru dat a na jeho základě i volba vhodné data-miningové metody. V praktické části pak nasazení vhodného data-miningového nástroje implementujícího tuto metodu, volba vhodné reprezentace dat a jejich úložiště, jakož i prezentace získaných výsledků.

Rozsah pracovní zprávy: **Dle pokynů vedoucího práce**
Rozsah grafických prací: **Dle pokynů vedoucího práce**
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

SHAFFER Clifford A.: A Practical Introduction to Data Structures and Algorithm Analysis Third Edition (Java). Virginia: Department of Computer Science, Virginia Tech, 2009 AGGARWAL, Charu C.: Data Mining – The Textbook. New York: Springer, 2015. ISBN 978-3-319-14142-8 SKALSKÁ H.: Data mining a klasifikační modely. Praha: Gaudeamus, 2010

Vedoucí diplomové práce: **doc. Ing. Tomáš Brandejský, Dr.**
Katedra softwarových technologií

Datum zadání diplomové práce: **5. listopadu 2019**
Termín odevzdání diplomové práce: **15. května 2020**

Univerzita Pardubice Česká republika
L.S.

Ing. Zdeněk Němec, Ph.D.
děkan

prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 15. listopadu 2019

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 7/2019 Pravidla pro odevzdávání, zveřejňování a formální úpravu závěrečných prací, ve znění pozdějších dodatků, bude práce zveřejněna prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 8. 8. 2020

Martin Bárta

PODĚKOVÁNÍ

Mé poděkování náleží doc. Ing. Tomáši Brandejskému, Dr. za znamenitou podporu v průběhu vypracovávání této diplomové práce. Dále bych chtěl poděkovat celé své rodině za podporu během celé délky mého studia.

ANOTACE

Předmětem této diplomové práce je představit způsob, jakým lze pracovat a analyzovat výsledky experimentů, jež generují velmi obsáhlé datové sady. Pro tuto úlohu jsou využity analytické nástroje pro velká data. Pomocí těchto nástrojů je zařízeno načtení dat, které je následováno hledáním užitečných informací a jejich následným zobrazením v přívětivé formě vizualizované pomocí tabulek a grafů. V teoretické části je vysvětlen proces a techniky dolovaní dat společně s představením nástrojů použitelných pro tento úkol. Praktická část pak obsahuje postup analyzování dat s pomocí nástroje Apache Spark u vzorové datové sady získané při běhu algoritmu symbolické regrese za pomocí nástroje Apache Spark.

KLÍČOVÁ SLOVA

Dolování dat, datová sada, velká data, vizualizace, datová analýza, Apache Spark

TITLE

Experiment Results Datamining

ANNOTATION

The main subject of this diploma thesis is to introduce a way how to work and analyze the results of experiments which generate very big data sets. For this purpose, are used Big Data analytics tools. By using these tools, it is possible to load the data which is followed by the searching for the useful information which is later visualized in form of charts and data tables. There is explanation of data mining process and its techniques together with the introduction of tools usable for this task. Practical part contains solution of analyzing data using the Apache Spark on the example data set which was obtained by running symbolic regression algorithm.

KEYWORDS

Data Mining, data set, Big Data, visualization, data analysis, Apache Spark

OBSAH

SEZNAM ILUSTRACÍ A TABULEK

SEZNAM ZKRATEK A ZNAČEK

API	Application Programming Interface
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DB	Database
DDL	Data Definition Language
DML	Data Manipulation Language
ETL	Extract, Transform, Load
GB	Gigabyte
HDFS	Hadoop Distributed File System
JAR	Java Archive
JDBC	Java Database Connectivity
JVM	Java Virtual Machine
LTS	Long Term Support
MB	Megabyte
ML	Machine Learning
ORM	Object Relational Mapping
RDD	Resilient Distributed Dataset
SBT	Simple Building Tool
SLF4J	Simple Logging Facade for Java
SQL	Structured Query Language
SSD	Solid-state Drive
TCP	Transmission Control Protocol
UDF	User Defined Function
URL	Uniform Resource Locator

ÚVOD

Potřeba analyzovat velké datové sady se neustále zvyšuje, stejně jako objem dat samotných. Pro praktický případ využití netřeba chodit daleko, ať už jde o analyzování chování spotřebitelů v rámci online služeb či třeba předpovídaní událostí na základě dlouhodobě shromažďovaných dat například o počasí, jedno zůstává u těchto úloh společné, a to je potřeba ohromných systémových prostředků. Čím vetší objem dat, tím vetší potřeba systémových prostředků, popřípadě času, který je třeba vynaložit pro zpracování velkého množství dat.

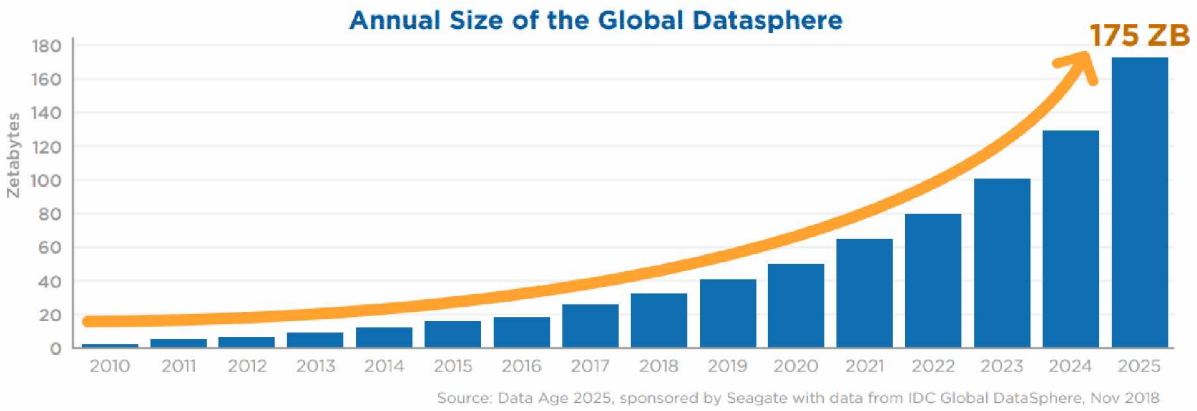
V současnosti se těmto problémům věnují celé týmy inženýrů v rámci zainteresovaných firem. Pro tyto potřeby vzniklo v průběhu času mnoho nástrojů, které práci s velkými daty umožňují a všemožně ji standardizují a usnadňují. V teoretické části této práce jsou některé tyto nástroje pro analýzu dat představeny, stejně tak jsou zde popsána některá úskalí a překážky na které může badatel narazit. Tato část pak v ideálním případě poslouží k nalezení vhodného nástroje pro analýzu vzorové datové sady. Hlavním zvoleným nástrojem pro analýzu dat se stal framework Apache Spark, jehož základní funkce jsou rovněž popsány v teoretické části.

Tato analýza je cílem praktické části této práce. K nalezení tohoto nástroje poslouží především informace získané z dostupných zdrojů, ať už z knih, nebo z internetových článků. Cílem práce je tedy nalézt a úspěšně použít jeden či kombinaci vhodných nástrojů, které umožní, pokud možno co nejpohodlněji, pracovat s dodanými daty. Nalezené řešení je skutečně kombinace více nástrojů a integrace mezi těmito představuje skutečnou výzvu, především pak pro někoho, bez předchozích zkušeností, ať už s analýzou dat nebo prací s velkými daty.

V praktické části je použit výhradně volně dostupný software, což v problematice analýzy dat rozhodně není limitující, neboť je k dispozici celá řada open-source nástrojů pro tyto účely. Konečným výstupem praktické části jsou vizualizace ve formě grafů, které v případě úspěchu nabízejí badateli zcela nový pohled na vstupní data. Nicméně je třeba mít na paměti, že tato práce není ani tak o výsledcích jako o cestě, která k těmto výsledkům vede. Cesta, jež může být, jak již bylo řečeno, pro člověka bez předchozích znalostí nástrojů a technik dolování a analýzy dat značně zdlouhavá a trnitá.

1. DOLOVÁNÍ DAT

Potřeba analyzovat data nepochybně předchází rozvoji informačních technologií, nicméně až teprve s příchodem prvních počítačů tato disciplína nabyla zcela jiných rozměrů. Objemy strojově zpracovaných dat, rychlost a efektivita, to vše se právě díky počítačům násobně zvýšilo. V průběhu času se stalo dolování dat velice ceněnou a žádanou dovedností. Objem shromažďovaných dat, která mohou následně sloužit k dolování dat neustále narůstá. Vždyť některé z největších technologických firem mají přímo na analýze dat založenou podstatnou část svých příjmů. Data, která je třeba analyzovat však mohou pocházet odkudkoli. Ať už jde o různé vědecké pokusy, či historické záznamy událostí, v podstatě všechna data lze podrobit analýze. Nárůst celosvětově vyprodukovaných dat je patrný na grafu níže. [1] [2] [3]

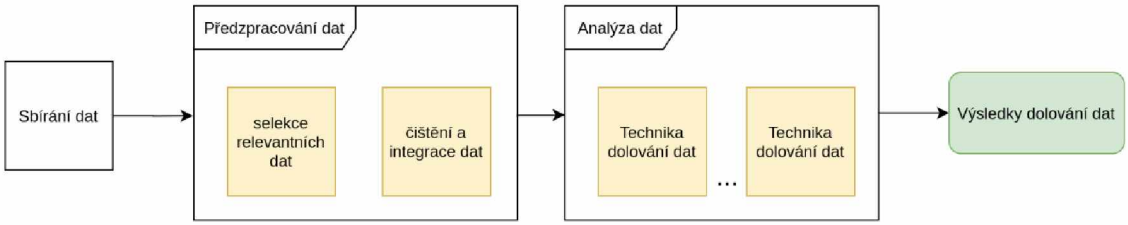


Obrázek 1 – Nárůst obejmu vyprodukovaných dat v čase s předpovědí [1]

Na pojem dolování dat lze nahlížet z několika různých pohledů. Většinou však označuje soubor činností, které zahrnují samotné získání dat, jejich následné čistění, zpracovávání, analyzování a následné získání užitečných informací. V podstatě jde o snahu najít nepředpokládané nebo dříve nám neznámé vztahy ve zkoumané datové sadě. Jedná se o náročnou disciplínu, jež v počítačovém světe využívá technologií jako strojové učení, statistiku, umělou inteligenci, databázové systémy atd. [3]

1.1 Proces dolování dat

Na obrázku níže je znázorněn zjednodušený proces dolování dat.



Obrázek 2 – Proces dolování dat [3]

Dolování dat začíná vždy sběrem dat, ať už se jedná o výsledky experimentu nebo například databázové záznamy o nákupech uživatelů v online obchodu, vždy je potřeba se k datům dostat. Vstupní data můžeme dostat najednou, popřípadě se může jednat o tzv. streamovaný vstup, kde zpracování dat probíhá takřka v reálném čase při vzniku části dat neustále. [3]

Následuje fáze předpřipravení dat. Součástí této fáze jsou obvykle dva kroky, a to selekce relevantních dat po které následuje čistění a integrace dat. Selekce dat může být zjednodušeně vysvětlena jako extrakce pro analýzu relevantních dat. Vstupní data mohou často pocházet z více zdrojů a mohou mít i zcela odlišné formáty. Pro příklad můžeme uvést aplikaci, která má jak databázové uložiště, tak textové soubory, kde se ukládají logové záznamy z běhu aplikace. V praxi pak můžeme sledovat třeba úbytek uživatelů při problému v běhu aplikace a podobně. Logy společně s aktivitou uživatelů mohou přispět k opětovnému nasimulování problému v aplikaci atd. Abychom ale mohli pohodlně využít techniky dolování dat, musíme s těmito informacemi pracovat najednou a efektivně. [3]

Po provedení selekce dat následuje čištění dat. V této fázi se buďto odstraňují záznamy, které jsou neúplné, nebo se mohou data vhodně doplnit. To záleží na jednotlivých případech, zda jsme schopni tyto informace doplnit, popřípadě zda je to z hlediska času žádoucí. Integrace dat často znamená vzít vybraná data a uložit je do formátu vhodného pro následující analytické zpracování. Podporované formáty bývají odlišné napříč jednotlivými nástroji pro analýzu dat. Často se může měnit i struktura dat, neboť některé metody například dobře nepracují s daty, která mají hodně dimenzí apod. [3]

Samotná analýza spočívá v aplikování vhodně zvolených metod. Hledaní odpovědí na otázky, pro které chceme znát odpověď, popřípadě hledat závislosti o kterých zatím nevíme. Největší výzva spočívá v tom, že každé dolování dat je unikátní. Neexistuje žádný univerzální a ověřený postup. Je jen na badateli, jaké zvolí metody a jestli následně tyto metody přinesou relevantní

výsledky. Klíčové faktory, které často rozhodují o úspěchu a neúspěchu, jsou zkušenosti a schopnosti badatele. Dolování a analýzu dat se lze naučit pouze praxí. [3]

Výsledky dolování dat mohou být různé. Někdy postačí pouhá odpověď na otázku v podobě prostého "ano či ne", často jsou pak výstupem různé grafické vizualizace nebo nová, podstatně zmenšená datová sada.

1.2 Některé problémy spojené s dolováním dat

- Jen skuteční odborníci mohou formulovat dotazy pro dolování dat.
- Často se při dolování dat používá menší datová sada pro trénink, na které se odzkouší daná technika, která poté nemusí fungovat správně u větší datové sady.
- Když některá data chybí, musí se často upravit i zdroj shromažďování dat tak, aby potřebná data byla přítomna.
- Pokud není datová sada dostatečně obsáhlá vyvozené závěry mohou být chybné.
- Dolování dat často vyžaduje obsáhlou databázi, která je náročná na obsluhu. [4]

1.3 Dolování dat a velká data

Ačkoliv dolování dat a samotný pojem velká data označují dvě různé disciplíny počítačových věd, v praktickém případě budeme dolovat data z velké datové sady, a tak nám tyto pojmy mohou splynout. Zatímco velká data neboli anglicky Big Data se zabývají jen možnostmi, jak efektivně pracovat s velkými daty, samotné dolování dat a analýza dat je pak to, co datům dává smysl a využití. Ve zkratce dolování dat nemusí souviset s velkými daty, neboť se techniky dají použít na jakkoli velkou datovou sadu, na druhou stranu ale termín velká data úzce souvisí s dolováním dat, neboť jen tehdy, pokud jsme schopni velkou datovou sadu analyzovat má cenu vůbec data shromažďovat. Data lze obvykle označit za velká, pokud je nelze dostatečně rychle zpracovávat pomocí relačních databází. [3] [5]

2. POUŽITÉ TECHNOLOGIE

Při implementaci analýzy dat u vzorového příkladu bylo ve společné kombinaci použito několik technologií. Jejich funkce a použité verze jsou k dispozici v tabulce níže.

Technologie	Použití	Verze
Spark	Zpracování a analýza dat.	2.4.5
Scala	Programovací jazyk požitý pro program, který nahrává data do DB. Zpracování velkého množství dat za pomocí frameworku Spark.	2.11.12
SBT	Nástroj sestavení pro Scalu.	1.3.8
SBT-Assembly	SBT plugin pro vytváření spustitelného souboru, který obsahuje externí knihovny.	0.14.10
MariDB Java Client	JDBC ovladač pro MariaDB použitelný ve Scale.	2.5.2
Skinny Framework – Skinny-ORM	Objektově relační mapování pro jazyk Scala.	3.0.3
ScalikeJDBC	JDBC wrapper zjednodušující práci s DB ve Scale.	3.4.0
Java	Scala a Spark potřebuje JVM pro svůj běh.	1.8.0_242
Python	Jazyk pro zpracovaní zredukovaného množství dat za pomocí frameworku Spark.	3.7.4
Bokeh	Python knihovna pro vizualizaci výsledků.	1.3.4
Bkzep	Python knihovna pro napojení výstupu Bokeh vizualizace do Zeppelin sešitu.	0.6.1
Pandas	Python knihovna pro zpracování a analýzu dat, zde použito jen pro snadnější napojení Python Spark objektů do knihovny Bokeh.	0.25.1
Anaconda	Sada nástrojů pro práci daty.	2019.10
Zeppelin	Sešitové rozhraní pro přehlednou práci s více technologiemi naráz, zároveň umožňuje prezentaci výsledků.	0.8.2, 0.9.0-preview1
MariaDB	Databázový server pro uložení extrahovaných dat experimentů.	10.4.12

Tabulka 1 – Použité technologie pro praktický příklad

2.1 Programovací jazyk Scala

Scala je multiparadigmatický programovací jazyk. Podporuje psaní programů, které mohou být jak funkcionální, tak objektově-orientované, popř. jejich kombinací. Název Scala je zkratka z anglického "scalable language", neboli škálovatelný jazyk v překladu. Jazyk vznikl v roce 2003, kdy se snažil adresovat některé neduhy jazyka Java. Koncepčně má umožnovat rychlejší vývoj aplikací. Jedná se o JVM jazyk, což v praxi to znamená, že veškerý Scala kód je kompilován do Java bytecode, který poté běží v JVM. Tento přístup má výhodu a tou je možnost používat knihovny z jazyka Java, kterých je nepřeberné množství. [6]

2.1.1 SBT

SBT je volně dostupný nástroj pro sestavení aplikací. Defaultně podporuje jazyk Scala a je tak alternativou k nástroji Maven používaným pro Javu. Podporuje projekty, které obsahují jak Java, tak Scala kód dohromady. Pro pohodlnou a intuitivní práci nabízí SBT interaktivní konzoli, stejně tak umožňuje napojení na Maven uložiště, kde mohou být uloženy jak Scala tak Java knihovny. Stejně jako Maven má SBT možnost rozšířit svou funkcionalitu o nejrůznější pluginy. V praktické části je použit plugin Assembly, který umožňuje generovat jeden JAR soubor, v němž je obsažen zdrojový kód aplikace společně se všemi knihovnami. Tento přístup usnadňuje práci a následnou distribuci aplikace. [7]

2.2 Programovací jazyk Python

Programovací jazyk Python je oblíben u vědců a matematiků, díky tomu existuje pro Python velké množství knihoven pro účely jako tvorba grafů či dolování dat. Python je jako programovací jazyk na špici, co se týče oblasti datových věd. Se svojí obrovskou uživatelskou základnou datových vědců nabízí taky skvělou podporu, co se týče dostupných materiálů a rad od komunity. Relativní jednoduchost dělá z Pythonu skvělou volbu pro někoho, kdo se učí programovat, což taky vedlo k masovému rozšíření jazyka Python. Python je v současnosti jazyk, u kterého počet uživatelů stoupá nejvíce ze všech programovacích jazyků. [8] [9]

2.3 Sešitové rozhrání Apache Zeppelin

Sešitové rozhraní Apache Zeppelin umožňuje vytvářet tzv. Sešity. Tyto sešity poté umožňují zjednodušenou centralizovanou práci s mnoha podporovanými technologiemi. K využití té či oné technologie slouží interpret. Interpret je Zeppelin plugin, který umožňuje uživateli využít jakýkoliv jazyk či data zpracovávající nástroj. Autoři Zeppelinu s komunitou vytvořili celou řadu již implementovaných interpretů pro nejrůznější technologie. Jejich seznam je k nahlédnutí na stránkách výrobce, odkud pochází i následující obrázek. [10]

Zeppelin	0.8.2	0.8.1	0.8.0	0.7.3	0.7.1 - 0.7.2	0.7.0	0.6.2 - 0.6.1
Spark	1.5.x, 1.6.x, 2.0.x, 2.1.x, 2.2.x, 2.3.x, **2.4.0**	1.5.x, 1.6.x, 2.0.x, 2.1.x, 2.2.x, **2.3.1**	1.4.x, 1.5.x, 1.6.x, 2.0.x, 2.1.x, **2.2.0**	1.4.x, 1.5.x, 1.6.x, 2.0.x **2.1.0**	1.4.x, 1.5.x, 1.6.x, 2.0.x **2.1.0**	1.1.x, 1.2.x, 1.3.x, 1.4.x, 1.5.x, 1.6.x, **2.0.0** Support Scala 2.11	1.1.x, 1.2.x, 1.3.x, 1.4.x, 1.5.x, 1.6.x SparkR is available
JDBC	PostgreSQL, MySQL, MariaDB, Redshift, Hive, Phoenix, Drill, Tajo are available	PostgreSQL, MySQL, MariaDB, Redshift, Hive, Phoenix, Drill, Tajo are available	PostgreSQL, MySQL, MariaDB, Redshift, Hive, Phoenix, Drill, Tajo are available	PostgreSQL, MySQL, MariaDB, Redshift, Hive, Phoenix, Drill, Tajo are available	PostgreSQL, MySQL, MariaDB, Redshift, Hive, Phoenix, Drill, Tajo are available	PostgreSQL, MySQL, MariaDB, Redshift, Hive, Phoenix, Drill, Tajo are available	PostgreSQL, MySQL, MariaDB, Redshift, Hive, Phoenix, Drill, Tajo are available
Pig	O	O	O	O	O	N/A	N/A
Beam	O	O	O	O	O	N/A	N/A
Scio	O	O	O	O	O	N/A	N/A
BigQuery	O	O	O	O	O	O	N/A
Python	O	O	O	O	O	O	O
Livy	O	O	O	O	O	O	O
HDFS	O	O	O	O	O	O	O
Alluxio	O	O	O	O	O	O	O
Hbase	O	O	O	O	O	O	O
Scalding	O	O	O	O	O	O	O
Elasticsearch	O	O	O	O	O	O	O
Angular	O	O	O	O	O	O	O
Markdown	O	O	O	O	O	O	O
Shell	O	O	O	O	O	O	O
Flink	O	O	O	O	O	O	O
Cassandra	O	O	O	O	O	O	O
Geode	O	O	O	O	O	O	O
Ignite	1.9.0	1.9.0	1.9.0	1.9.0	1.7.0	1.7.0	1.6.0
Kylin	O	O	O	O	O	O	O
Lens	O	O	O	O	O	O	O
PostgreSQL	O	O	O	O	O	O	O

Obrázek 3 – Seznam interpretů pro jednotlivé Zeppelin verze [10]

Mezi další funkce tohoto sešitového rozhraní patří například integrovaná podpora pro tvorbu grafů. Graf lze vytvořit například z vráceného databázového objektu, popřípadě je zde například podpora vizualizace přímo ze Spark objektu, jako je například Spark DataFrame, ať už jde o Scala Spark nebo o Python Spark verzi. Tato integrovaná funkce je sice dobře vypadající, ale neumožňuje až tolik customizace a volnosti jakou bychom od vizualizačního nástroje mohli potřebovat, nicméně i s tím autoři počítali, a Zeppelin sešity tak umožňují i práci s více standardizovanými a populárnímu knihovnami na vytváření grafů a dalších vizualizací. Můžeme tak například použít oblíbenou knihovnu Matplotlib nebo třeba Bokeh knihovnu, která bude využita v praktické části této práce. [10]

Další výhodou nástroje jako je Apache Zeppelin je do jisté míry zjednodušení práce s danými technologiemi. V případě Apache Spark interpretu například nemusíme vytvářet SparkSession ani manuálně ukončovat Spark, to vše za nás udělá příslušný interpret. Práce se Zeppelinem je podrobněji přiblížena v praktické části. [10]

2.4 Framework Spark

Apache Spark je obecný zpracovávač dat. Je určen pro výpočty v klastru, ale může běžet i na jednom počítači. Od počátku byl Spark optimalizován tak, aby běžel, pokud možno celý v operační paměti. To umožňuje zpracovávat data mnohem rychleji než jeho předchůdce Hadoop. V některých případech se uvádí až stonásobně. Ovšem i v případě, že musí Spark uložit nebo načíst data z disku, může být i desetkrát rychlejší než Hadoop MapReduce metoda. Abychom ale zůstali fér, Spark úplně nenahrazuje Hadoop. Pro některé typy úloh se Hadoop stále hodí více, v některých případek pak lze s výhodou využít jejich kombinaci. Abychom zjistili, kdy použít Hadoop a kdy Spark je v samostatné Spark kapitole k dispozici srovnání těchto dvou technologií. [11] [12]

2.5 MariaDB

MariaDB Server je jedna z nejpopulárnějších open-source relačních databází. Vývoj MariaDB má na starosti původní tým, který stojí za MySQL databází. MariaDB přímo z MySQL vychází, v jednom bodě se z MySQL oddělila a od té doby vývoj pokračuje nezávisle. MariaDB vznikla jako odpověď na akvizici MySQL firmou Oracle. Cílem je udržet toto databázové řešení open-source a také neustále rozšiřovat funkčnost. MariaDB databázové řešení používá například Google, Mozilla nebo třeba DBS Bank. V roce 2017 podpořila vývoj MariaDB 25 miliony eur i Evropská investiční banka. [13]

2.6 Bokeh

Bokeh je knihovna nabízející interaktivní vizualizace v moderních webových prohlížečích. Knihovna nabízí nástroje pro tvorbu interaktivních grafů, ale i jiných vizualizací, umožňuje zpracování obsáhlých dat. Knihovna se dá rovněž napojit na sešitové rozhrání Zeppelin, v takovém případě se grafický výstup promítá přímo do sešitu. Knihovna je postavená na jazyku Python, samotné vykreslování do prohlížeče je prováděno jazykem Javascript. V případě interaktivity nad rámec Javascriptu, typicky když je potřeba na akci uživatele reagovat v Jazyce Python, jako například načíst jiná data a podobně, vyžaduje Bokeh aplikace svůj aplikační server. [14]

3. SPARK

Spark je nástroj určený k distribuovaným výpočtům, původně vznikl v roce 2009 jako univerzitní projekt na půdě Kalifornské univerzity. V tomto čase výzkumníci pozorovali neefektivitu, s kterou metoda MapReduce frameworku Hadoop nakládá s daty pří určitých případech. Jako odpověď na to vznikl nový přístup, který provádí, pokud možno, všechny operace s daty v operační paměti. Z tohoto důvodu je Spark opravdu rychlý, nicméně je potřeba mít této paměti dostatek. Tato paměť je mnohonásobně dražší než diskové uložiště. V případě nedostatku operační paměti pro danou úlohu Spark přesouvá některá data na disk, což samozřejmě prodlužuje dobu, za kterou je úloha zpracována. Dalším zlepšení, s kterým výzkumníci přišli je nový přístup k zaručení odolnosti proti výpadkům a chybám. V případě Hadoopu jsou ukládány všechny mezikroky, které vedou k výsledku na disk. V případě Sparku se ukládá jen záznam provedených operací a v případě výpadku se znova tyto operace provedou. Toto řešení může být v případě výpadku pomalejší, ale samotné zpracovávaní je kvůli minimalizovanému záznamu dat na disk rychlejší. [11] [12]

V roce 2013 se Spark stal inkubačním projektem softwarové nadace Apache. V roce 2014 se pak tento projekt stává jedním z nejdůležitějších projektů v portfoliu Apache. Vývoj Sparku probíhá komunitně, s podporou známých technologických firmem jako například IBM, Databricks, Google, Facebook a mnoha dalších. [11]

3.1 Na co vše se Spark používá

- Zpracování dat v reálném čase – Spark umožňuje zpracovávat příchozí data bez předchozího uložení na disk. Data mohou chodit najednou z více zdrojů, samozřejmě, že by data mohla být uložena na disk a poté zpracována, ale v některých případech to z bezpečnostních důvodů není možné, jako například u záznamů z bankovních transakcí, kde by každé uložení na disk představovalo zbytečné riziko.
- Strojové učení – Schopnost Sparku pracovat s daty v operační paměti je ideální pro časté a opakované dotazy, což se hodí v algoritmech strojového učení. Spark dokonce obsahuje svoji knihovnu zaměřenou právě na strojové učení a tou je MLlib. Tato knihovna obsahuje obecné učící algoritmy jako například klasifikaci, regresi a další.
- Interaktivní analýza – Spark dokáže rychle reagovat na změny dotazů. Místo vykonávání předem definovaných dotazů, tak lze snadno Sparku podsunout konkrétnější dotaz pomocí interaktivních dotazů například v interaktivní Spark konzoli.

- Integrace dat – Rychlost, s jakou Spark zpracovává velké množství dat z něj dělá ideální nástroj pro předzpracování velkého množství dat, a to i z více zdrojů. Obecně tuto integraci dat můžeme shrnout pod zkratkou ETL. Z anglického extract, transform a load. V překladu extrahuj, transformuj a načti. Celý proces si lze představit tak, že na vstupu máme velké množství dat, transformacemi tyto data očistíme, poté provedeme například zjednodušení pomocí statistických ukazatelů a tyto zjednodušená data dáme k dispozici k dalšímu zpracování či vizualizaci a podobně. [11]

3.2 Spark aplikace

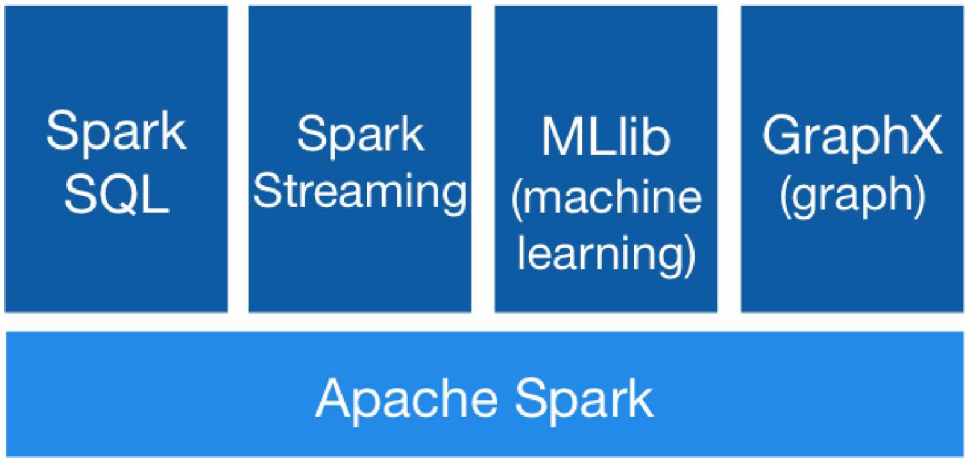
Spark aplikace se skládá ze dvou částí. První část je aplikace pro zpracování dat, skládá se z logiky vyjádřené za pomocí Spark API. Druhá část je Spark ovladač neboli driver anglicky. Zatímco první část může být jen pár řádek kódu, druhá část se nemění. Spark ovladač je v podstatě centrální koordinátor, který interaguje s cluster manažerem s pomocí kterého Spark rozhodne, na kterém stroji, popřípadě strojích bude probíhat zpracování dat. Spark ovladač požádá cluster manažer, aby na každém stroji spustil proces s názvem Spark executor, v překladu vykonavatel. Spark ovladač poté servíruje jednotlivé úlohy jednotlivým vykonavačům. Další úloha ovladače je spojení a prezentace výsledů uživateli. Vstupním bodem do Spark aplikace je SparkSession, pomocí tohoto objektu lze Spark aplikace různě konfigurovat, stejně tak nabízí API pro vyjádření logiky zpracování dat. [11] [12]

3.2.1 Spark ovladač a vykonavač

Každý Spark vykonavač je JVM proces alokovaný exklusivně pro danou Spark aplikaci. Tato exkluzivní alokace má zamezit přenášení nežádoucího chování napříč Spark aplikacemi, které se tak nemohou ovlivňovat, v případě jedné aplikace, která se chová nežádoucím způsobem, zůstává toto chování v rámci jedné aplikace. Životnost Spark vykonavače je po dobu běhu Spark aplikace. Spark využívá model Master/Slave. V praxi to znamená, že Spark ovladač je master, a Spark vykonavač je slave. Každý z těchto procesů je spuštěn jako nezávislý proces v clusteru. Vykonavač ve své slave roli dělá vždy to, co mu ovladač z pozice mastera řekne, což je zpracování dat ve formě jednotlivých úkolů. Každý úkol je zpracováván pomocí jednoho CPU jádra, popř. vlákna. To je důvod proč Spark může snadno škálovat a urychlit proces zpracování velkého množství dat za pomocí paralelismu. [11] [12]

3.3 Spark struktura

Apache Spark nabízí celou řadu funkcionalit. Základem je jádro, na obrázku níže znázorněno jako blok Apache Spark.



Obrázek 4 – Struktura frameworku Spark [15]

Jádro obsahuje veškeré funkce k běhu distribuovaných programů jako plánování, koordinaci a chybovou korekci. Dále obsahuje generickou programovací abstrakci pro zpracování dat nazvanou Resilient Distributed Dataset, v překladu odolná distribuovaná datová sada, zkráceně pak RDD. Na obrázku nad jádrem jsou dále znázorněny další knihovny, které nabízejí ještě více specifičtější funkce jako doplněk k samotnému jádru. Tato jasná struktura a fakt, že Spark obsahuje prakticky veškerou možnou funkcionalitu, kterou lze pro zpracování a analýzu dat chtít v přehledném a jednotném API, dělá ze Sparku ideální nástroj na tvorbu aplikací, které jsou poměrně jednoduché na vývoj a nasazení. Další výhodou je možnost kombinovat funkcionalitu jednotlivých modulů. Můžeme tak mít aplikaci, která podporuje jak dávkové, tak streamované zpracování dat. V neposlední řadě pak Spark umožňuje vývoj aplikací, který dříve nebyl možný, například aplikaci, která umožňuje interaktivní dotazy do právě spočítaných výsledků strojového učení z dat, která přicházejí v reálném čase. Zjednodušeně si můžeme toto představit jako mobilní telefon s funkcemi jako kamera, volání a GPS, kdy vhodnou kombinací těchto funkcí lze vytvořit aplikaci jako je Waze, interaktivní navigaci s přehledem o dopravní situaci v reálném čase. [11] [12]

3.3.1 Spark jádro

Jak již bylo řečeno Spark jádro je základním kamenem každé Spark aplikace. Obsahuje dvě části. Část první se zabývá samotným distribuovaným chováním a druhá část pak obsahuje programovou abstrakci pomocí RDD. Infrastruktura distributivního zpracování je zodpovědná za distribuci a koordinaci, stejně tak za plánování jednotlivých úloh s daty napříč jednotlivými stroji v clusteru. Dvě hlavní úlohy, které musí distributivní zpracování řešit je samotné zpracování dat a zvládání chybových scénářů, tak aby například při výpadku jednoho stroje byla jeho úloha dokončena na jiném stroji a podobně. Tyto dvě základní funkcionality jsou známy také

pod pojmem "data shuffiling", neboli přesouvání dat. Pokročilí Spark uživatelé se vyznačují znalostí přesouvání dat a jsou schopni navrhovat aplikace s vysokým výkonem. [11]

Klíčem k abstrakci ve Sparku je již zmíněný koncept RDD. RDD API je k dispozici uživateli ve třech programovacích jazycích. Konkrétně jsou to Scala, Java a Python, a je jen na uživateli, který jazyk si zvolí. [11] [12]

3.3.2 Spark SQL

Spark SQL je modul postavený nad Spark jádrem. Jeho hlavním úkolem je škálovatelně pracovat se strukturovanými daty. Výhodou SQL modulu je lehkost, s jakou si uživatelé znalí jazyka SQL mohou osvojit práci s tímto modulem, neboť většina uživatelů Sparku zná SQL z prostředí relačních databází. Uživatel může pracovat s daty ohromné velikosti stejně jako kdyby pracoval a konstruoval databázové dotazy. K dispozici je také abstraktní DataFrame API pro více specifické úkoly. DataFrame je v podstatě distribuovaná kolekce organizovaná do sloupců. Zjednodušeně lze říčí, že DataFrame je ekvivalent tabulky, kterou známe z relačních databází. [11] [12]

Funkcí modulu Spark SQL je také načítaní dat z mnoha různých zdrojů jako jsou JSON, CSV, ORC nebo třeba parquet soubory, relační databáze, Hive tabulky, textové soubory a další. Schopnost pracovat s daty z různých zdrojů umožňuje Spark používat jako konvertor dat z jednoho zdroje do druhého. Podle Spark průzkumu z roku 2016 je Spark SQL nejrychleji se rozvíjející Spark modul, je to logické, vzhledem k tomu, že tento modul skutečně umožňuje nejenom skutečným odborníkům na datové vědy, ale taky osobám se znalostí SQL objevovat svět distributivního zpracování dat. Spark SQL moto je psát méně kódu, číst méně dat a nechat integrované optimalizace dělat těžkou práci. [11]

3.3.3 Spark Streaming

Spark Streaming je dalším Spark modulem. Tento modul se zabývá zpracováním dat v reálném čase. Modul je důležitý pro zpracování dat, která je potřeba vyhodnotit ihned, data, která nemohou zastarat, data, která při okamžitém vyhodnocení dávají uživateli konkurenční výhodu a podobně. Typickým příkladem může být třeba vývoj cen na burze. Stejně jako modul Spark SQL i modul Spark Streaming umí pracovat s daty z různých zdrojů, v tomto případě se jedná například o Flume, Kinesis, Twitter, HDFS, Kafka nebo TCP sokety. [11]

V dřívějších verzích Sparku bylo zpracování streamovaných dat řešeno přes DStream, což byla implementace inkrementovaného streamovaného zpracování. Data se rozdělila na malé dávky, každá z těchto dávek byla zpracována jako jeden RDD. [11]

Od verze Spark 2.1 je zpracování streamovaných dat vystavěno nad Spark SQL. Tento krok umožnil vývojářům ještě rychleji proniknout do zpracování dat v reálném čase, neboť nemusejí řešit, zda jde o dávková nebo streamovaná data, a k oběma typům se mohou chovat stejně. [11]

3.3.4 Spark MLlib

Spark MLlib modul obsahuje nástroje, které lze s výhodou využít při technikách strojového učení. Obsahem tohoto modulu je více než 50 běžných algoritmů strojového učení. Spark MLlib obsahuje dvě API, jedno vychází z RDD přístupu a druhé z DataFrame. Od verze 2.0 se RDD API upravuje a nedoporučuje se používat, zatímco API založené na DataFrame je doporučené. Toto API je navíc mnohem intuitivnější a práce s ním se podobá opět práci známé z SQL modulu. Jako další výhoda je pak uváděna zjednodušená práce s ML Pipeline, což je další API postavené nad DataFrame API, které umožňuje uživateli zjednodušené vytváření ML pipeline (zřetězené zpracování). [11] [12]

Algoritmy strojového učení jsou ze své podstaty iterativní, což znamená, že běží, dokud nejsou splněny podmínky zadaní. Spark umožňuje lehce implementovat takový algoritmus a poté ho spustit ve škálovatelném prostředí v clusteru. [11]

3.3.5 Spark GraphX

Grafové zpracování dat operuje se strukturami s vrcholy a hranami které je spojují. Datová struktura graf často reprezentuje objekty z reálného světa, jako například sociální sít LinedIn, nebo sít spojených webových stránek a podobně. GraphX je Spark komponenta, která podporuje zpracování paralelních výpočtů tím, že poskytuje abstrakci spojeného multigrafu s vlastnostmi přidruženými ke každému vrcholu a hraně. Komponenta rovněž obsahuje některé základní grafové algoritmy jako je hledání nejkratší cesty, PageRank pro ohodnocení webových stránek a další. [11]

3.4 Spark nebo Hadoop?

Hadoop a Spark jsou dva prominentní hráči, pokud jde o distribuované výpočty v clusteru, následující srovnání nám pomůže vybrat optimální nastroj pro praktickou úlohu této práce.

Jak již bylo zběžně nakousnuto v přehledu použitých technologií, který je součástí předchozí kapitoly, Spark je v podstatě odpověď na některé nedostatky, kterými Hadoop do jisté míry trpěl a trpí, není ovšem pravdou, že by vydáním Spark frameworku Hadoop zastaral a přestal se používat. Spark naopak s Hadoop frameworkem může skvěle spolupracovat a využít tak věcí, které samotný Spark neumí. Spark například umí pracovat s Hadoop cluster managerem nazývaným YARN, stejně tak se může napojit na distribuované Hadoop uložiště nazývané

HDSF. Samozřejmě Spark umí běžet také nezávisle, popřípadě se integrovat ještě s jinými technologiemi, než je Hadoop, jako příklad poslouží cluster manager Mesos, nebo sloupcové uložiště Cassandra. Je nicméně pravda, že s přibývající funkcionalitou Sparku a faktem, že se jedná o modernější a výkonnější řešení, nedává moc smysl tzv. na zelené louce zvolit Hadoop namísto Sparku. [11] [15]

Pro porovnání technologií Spark a Hadoop byly zvoleny následující kategorie: rychlost, cena, odolnost proti výpadkům, módy zpracování dat, bezpečnost a v neposlední řadě také podpora strojového učení. Následující kategorie jsou shrnuty v rychlém přehledu v tabulce níže, poté následuje detailnější popis některých kategorií. [16] [17]

Kategorie	Spark	Hadoop
Rychlost	Rychlý, umožňuje práci s distribuovanými daty a zpracování v reálném čase.	Není zaměřen na rychlost ale na obrovská distribuovaná data.
Snadnost použití	Obsahuje interaktivní mód, který zobrazuje výsledky dotazů a zpětnou vazbu.	MapReduce nemá interaktivní mód.
Cena	K dispozici zdarma i ke komerčnímu použití.	K dispozici zdarma i ke komerčnímu použití.
Cena hardwaru	Dražší, protože pro rychlý chod potřebuje více operační paměti.	Levnější, cena za rychlé diskové uložiště je menší než za operační paměť.
Zpracování dat	Umožňuje zpracovávat data dávkově i v reálném čase.	Pouze dávková data.
Odolnost proti výpadkům	RDD běží paralelně a v případě výpadku jsou data přepočítány ze záznamu transformací.	Může výrazně prodloužit čas operací, protože jsou data replikována na více strojích.
Bezpečnost	Zabezpečeno heslem, popřípadě se dá využít integrace s Hadoop a přes něj využít jiné druhy zabezpečení.	Podporuje Kerberos a další řešení třetích stran jako třeba LDAP.
Strojové učení	Má vlastní knihovnu na strojové učení.	Měl vlastní knihovnu na strojové učení, ale už není podporována, protože se MapReduce na to nehodí.

Tabulka 2 – Porovnání vlastností Spark a Hadoop [16] [17]

3.4.1 Rychlost

Spark s dostatkem operační paměti běží až stokrát rychleji a desetkrát rychleji s pevným diskem. Spark byl například použit pro setřídění 100 TB dat, která setřídil třikrát rychleji než Hadoop. Třikrát rychleji nezní jako desetkrát nebo stokrát, ale Spark pro tuto úlohu měl desetkrát méně výpočetních strojů. Veškerá data v testu byla uložena na disku v HDFS uložišti bez využití Spark cache paměti v operační paměti. Není tedy pochyb o tom, že Spark je skutečně mnohonásobně rychlejší, a to i v prostředí, které se dá označit jako Hadoop ideální. [16] [17] [18]

Spark převyšuje rychlostí Hadoop MapReduce metodu ze dvou důvodů. První je ten, že není vázán přístupem na pevný disk pokaždé, když se provádí část MapReduce metody. Druhý důvod jsou Spark DAG optimalizace před každým krokem. Hadoop nemá cyklické spojení mezi jednotlivými MapReduce kroky, a tudíž neprovádí optimalizace na rozdíl od Sparku. [16] [17]

Nicméně pokud Spark běží s YARN cluster managerem a sním i s dalšími službami na YARN napojenými, může výkonost Sparku klesat z důvodu přetížení RAM. Z tohoto důvodu může být Hadoop preferován nad Sparkem v případě dávkového zpracování dat. [11]

3.4.2 Cena

Jak Spark tak Hadoop jsou k dispozici zcela zdarma jako open-source řešení. Nicméně je třeba mít na paměti cenu údržby, cenu hardware, na kterém software poběží a v neposlední řadě také lidí, kteří se budou starat o cluster jako a budou zároveň rozumět jeho správě. Obecně platí, že Spark vyžaduje na strojích více operační paměti, zatímco Hadoop větší kapacitu diskového uložiště. Co se týče zaměstnanců, Spark jako novější systém pravděpodobně bude mít dražší a více žádané specialisty. Další možností, je si cluster pronajmout, v takovém případě je cena údržby přímo v ceně pronájmu. Společnosti, které se tímto zabývají jsou například DataBricks pro Spark nebo Claudera pro Hadoop. [17]

Porovnávat cenu clusteru optimálního pro Spark a Hadoop je obtížné, obecně lze prohlásit, že cluster pro optimální běh Sparku bude pravděpodobně dražší než cluster optimální pro Hadoop, nicméně je třeba vzít v potaz, že úloha, která bude zpracována za pomoci Sparku pravděpodobně doběhne dříve, a v tom případě i přes dražší běh clusteru může zpracování této úlohy pomocí Sparku vycházet levněji. [17]

3.4.3 Odolnost proti výpadkům

Hadoop a Spark zaručují odolnost proti výpadkům rozdílně. Hadoop MapReduce metoda používá pro tento účel službu TaskTracker v kombinaci se službou JobTracker. Jestliže se

TaskTracker neohlásí JobTrackeru, operace jsou naplánovány jinému JobTrackeru. Toto může poměrně významně prodloužit dobu zpracování dat, neboť jsou data replikována na více uzlech. V případě ztráty uzlu se data mohou obnovit z ostatních uzlů. Spark na druhou stranu používá již zmíněné RDD, které mají zaručenou odolnost proti výpadkům rozdílněji. Protože RDD operuje paralelně odkazují do sdíleného uložiště. Díky tomu si RDD ukládají data v paměti jednotlivých strojů, pokud je RDD ztracen automaticky se dopočítá za použití původních transformací na jiném stroji s tím, že bohužel musí začít od začátku, neboť jsou uloženy jen záznamy o provedených operacích, a ne jejich výsledky.

Nelze jednoznačně říct, který přístup je v případě výpadku rychlejší, vždy záleží na rozsahu výpadku a na tom, kolik toho musí Spark znova dopočítat. Nicméně lze říct, že v případě běhu bez chyby je řešení Sparku výrazně rychlejší, neboť nemusí replikovat a ukládat data na více strojích. [11] [16]

3.5 Proč byl vybrán Spark?

Spark byl vybrán pro zpracování praktické části z několika důvodů. Data z experimentů se sice nevejdou do operační paměti počítače, na kterém byla zpracovávána, ale to vlastně ani tak moc nevadí, neboť si Spark může odložit data na pevný disk, a i tak by měl být pořád rychlejší nebo alespoň stejně rychlý jako Hadoop. Při velikosti dat, která jsou v řádech nižších stovek GB je jediné omezení to, aby se data vešla na disk najednou, tedy pokud chceme zpracovávat veškerá data najednou. Spark umí pracovat jak se sdílenými, tak distributivními zdroji dat. V případě praktické úlohy této práce je využita instance MariaDB běžící lokálně na stejném počítači jako Spark. Pokud bychom ale chtěli zpracovávat data v clusteru, což by byl typický příklad použití Sparku, bylo by možné popřemýšlet i nad výhodou využití distribuovaného uložiště. Pro toto řešení by mohl Spark fungovat i v kombinaci s Hadoop HDFS, což ale není náš případ, neboť nám stačí data zpracovat lokálně na jednom počítači. K důvodům jako je rychlost a snadnost nastavení pak musíme v neposlední řadě přidat i přítomnost interaktivního módu nabízející rychlou odezvu a moderní API. Spark je celkově modernější a uživatelsky přívětivější, a dokud se neobjeví na trhu další hráč, bude nejspíše Spark jasná volba. Jediný případ, kdy lze s výhodou použít Hadoop oproti Sparku je, pokud chceme zpracovat ohromné množství dat co nejlevněji a nejde nám o čas, za který mají být známé výsledky. [11]

3.6 Alternativy

Pokud bychom se zaměřili jen na distribuované zpracovaní dat, existují další méně používané nástroje.

Prvním zástupcem je Apache Storm, který slouží k zpracovávání dat v reálném čase, což je úloha, kterou Apache Spark zvládá taky. Pro zpracování praktické části této práce ovšem zpracování dat v reálném čase není potřeba. [19] [20]

Další z mnoha alternativ je Hydra, což je další systém pro distribuované zpracování dat, podobně jako Spark zvládá jak dávkové zpracování dat, tak data v reálném čase. [19] [20]

Jako poslední příklad lze uvést BigQuery od společnosti Google. Tato poměrně nová technologie je spíše nahlédnutí do budoucnosti, pokud se prosadí. Jedná se o systém, který podporuje SQL dotazy nezávisle nad uložištěm, nad kterým operuje. To je podobné jako u Spark SQL. Platforma obsahuje některé předpřipravené nástroje pro dolování dat. Výhodou i nevýhodou pak je, že veškerá data musí být nahrána na servery spravované samotným Googlem. Díky tomu dostaneme přístup k rychlému hardwaru, nicméně za cenu předaní dat třetí straně. Cena za službu se odvíjí podle velikosti dat. V případě velkého objemu dat může tato služba být velice drahá. V době psaní této práce bylo prvních 10 GB každý měsíc zdarma. [19] [20]

4. PŘÍPRAVA PROSTŘEDÍ PRO ANALÝZU VZOROVÝCH DAT

V několika následujících krocích bude provedena instalace nástrojů a technologií potřebných ke splnění zadání praktické části této práce. Kombinace těchto nástrojů zajistí vše, co je potřeba k analýze dat. Výsledné prostředí by mělo být intuitivní a uživatelsky přívětivé. Velké množství faktorů bylo vzato v úvahu a výsledná kombinace nástrojů spolu skutečně spolupracuje bez zjevných problémů. V ideálním případě zabere instalace následujících nástrojů několik hodin. Následující návod je detailní a měl by tak zkušenějšímu uživateli obeznámeným s operačním systémem Linux stačit na zprovoznění všech potřebných nástrojů.

4.1 Operační systém

Jako operační pro analýzu dodaných dat byl po předchozích marných experimentech s operačním systémem Windows 10 zvolen Linux, konkrétně pak Ubuntu distribuce ve verzi 18.04.4 LTS. LTS zkratka zde indikuje verzi s dlouhodobou podporou, jedná se o nejnovější LTS verzi na počátku psaní této práce. S tímto operačním systémem se nevyskytly žádné další integrační problémy, což se o operačním systému Windows bohužel nedá říct. Nepřekonatelný problém nastal s aplikací Apache Zeppelin a její integrace s vlastní verzí Apache Spark. Zatímco s Ubuntu systémem toto funguje, ve Windows funguje korektně na Zeppelinu pouze již se Zeppelin aplikací distribuovaný Spark. Nemůžeme se ovšem kvůli tomu na Zeppelin příliš zlobit, neboť Windows ve verzi 10 není zatím podporovaný, narozdíl od Windows 7. Samotná instalace operačního systému není předmětem této práce a není tak zde popsána. [10]

4.2 MariaDB

Jako databázové uložiště bylo zvoleno open-source řešení MariaDB. V podstatě by šla použít jakákoliv jiná databáze, do které se dá připojit pomocí JDBC, neboť přes toto rozhrání se lze v Apache Spark připojit do vybrané databáze. Rovněž by šlo využít i jiné uložiště, než je relační databáze.

Nainstalovat MariaDB server a klient lze v prostředí Ubuntu několika způsoby. Nejjednodušší je naistalovat MariaDB z uložiště Ubuntu. Nicméně toto uložiště ne vždy obsahuje aktuální verzi. Pokud nás verze víceméně nezajímá, postačí nám v terminálu zadat příkaz níže. [23]

sudo apt install mariadb-server

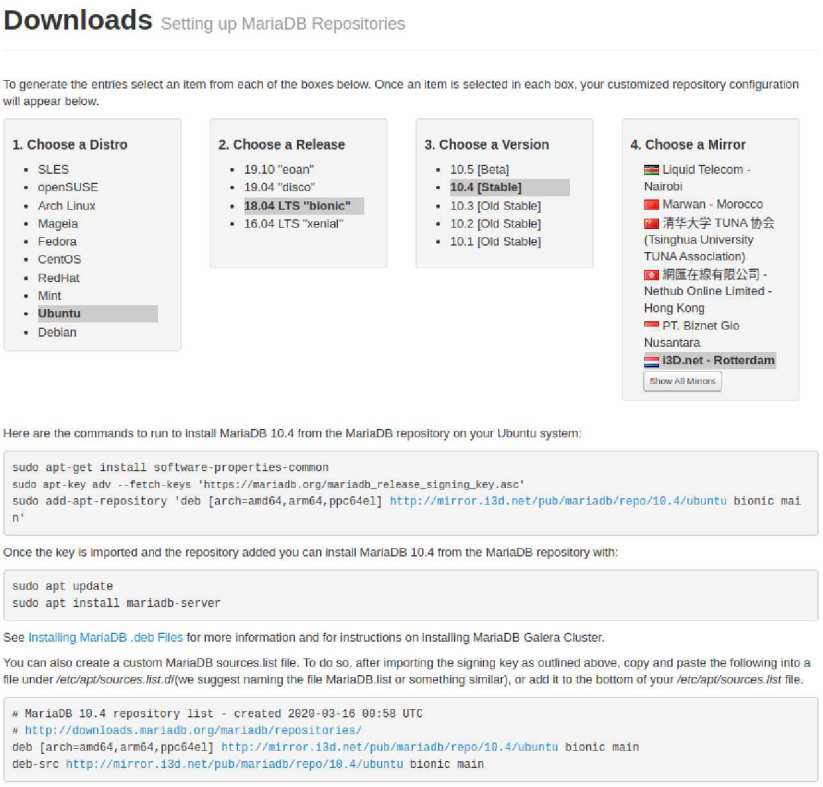
Jaká verze aplikace je k dispozici na defaultním uložišti lze zjistit příkazem níže.

apt policy mariadb-server

Po zadaní tohoto příkazu se zobrazí, kde se daná aplikace nachází a jaké verze jsou k dispozici. V době psaní této práce se do konzole vypsalo toto.

mariadb-server:
Installed: (none)
Candidate: 1:10.1.44-0ubuntu0.18.04.1
Version table:
1:10.1.44-0ubuntu0.18.04.1 500
500 http://cz.archive.ubuntu.com/ubuntu bionic-updates/universe amd64 Packages
500 http://cz.archive.ubuntu.com/ubuntu bionic-updates/universe i386 Packages
500 http://security.ubuntu.com/ubuntu bionic-security/universe amd64 Packages
500 http://security.ubuntu.com/ubuntu bionic-security/universe i386 Packages
1:10.1.29-6 500
500 http://cz.archive.ubuntu.com/ubuntu bionic/universe amd64 Packages
500 http://cz.archive.ubuntu.com/ubuntu bionic/universe i386 Packages

Z výpisu je patrné, že nejnovější dostupná verze v Ubuntu uložišti je 10.1.44. Pokud bychom ovšem chtěli novější verzi MariaDB, lze využít povedený nástroj na stránkách dokumentace https://downloads.mariadb.org/mariadb/repositories/#mirror=i3dnet. Na této internetové stránce můžeme po zvolení příslušného operačního systému vidět podporované verze MariaDB, které se nachází na uložišti výrobce databázového řešení, a to včetně příkazů pro nainstalování, jak lze vidět na obrázku níže. [22]



Obrázek 5 – Získání nejnovější verze MariaDB [21]

Po přeložení popisků vygenerovaných na stránce výrobce dostaneme představu, co dané příkazy vlastně dělají. Nejdříve se nainstaluje balíček software-properties-common. S tímto balíčkem můžeme vyčíst klíč a uložiště na kterém se nachází požadovaný software. Následně se uloží samotné uložiště do lokální databáze uložišť, ze kterých můžeme instalovat software. [22]

Pokud se povede přidat MariaDB uložiště do seznamu uložišť, bude výpis po opětovné spuštění příkazu "*apt policy mariadb-server*" podstatně delší, podobně jako níže.

mariadb-server:
Installed: (none)
Candidate: 1:10.4.12+maria~bionic
Version table:
1:10.4.12+maria~bionic 1000
500 http://sfo1.mirrors.digitalocean.com/mariadb/repo/10.4/ubuntu bionic/main amd64 Packages
500 http://sfo1.mirrors.digitalocean.com/mariadb/repo/10.4/ubuntu bionic/main arm64 Packages
500 http://sfo1.mirrors.digitalocean.com/mariadb/repo/10.4/ubuntu bionic/main ppc64el Packages
1000 http://downloads.mariadb.com/MariaDB/mariadb-10.4/repo/ubuntu bionic/main amd64 Packages
1:10.4.11+maria~bionic 1000
500 http://sfo1.mirrors.digitalocean.com/mariadb/repo/10.4/ubuntu bionic/main amd64 Packages
500 http://sfo1.mirrors.digitalocean.com/mariadb/repo/10.4/ubuntu bionic/main arm64 Packages
500 http://sfo1.mirrors.digitalocean.com/mariadb/repo/10.4/ubuntu bionic/main ppc64el Packages
1000 http://downloads.mariadb.com/MariaDB/mariadb-10.4/repo/ubuntu bionic/main amd64 Packages
1:10.4.10+maria~bionic 1000
500 http://sfo1.mirrors.digitalocean.com/mariadb/repo/10.4/ubuntu bionic/main amd64 Packages
500 http://sfo1.mirrors.digitalocean.com/mariadb/repo/10.4/ubuntu bionic/main arm64 Packages
500 http://sfo1.mirrors.digitalocean.com/mariadb/repo/10.4/ubuntu bionic/main ppc64el Packages
1000 http://downloads.mariadb.com/MariaDB/mariadb-10.4/repo/ubuntu bionic/main amd64 Packages
1:10.3.22+maria~bionic 500
500 http://sfo1.mirrors.digitalocean.com/mariadb/repo/10.3/ubuntu bionic/main amd64 Packages
500 http://sfo1.mirrors.digitalocean.com/mariadb/repo/10.3/ubuntu bionic/main ppc64el Packages
500 http://sfo1.mirrors.digitalocean.com/mariadb/repo/10.3/ubuntu bionic/main arm64 Packages
1:10.3.21+maria~bionic 500

500 http://sfo1.mirrors.digitalocean.com/mariadb/repo/10.3/ubuntu bionic/main amd64 Packages
500 http://sfo1.mirrors.digitalocean.com/mariadb/repo/10.3/ubuntu bionic/main ppc64el Packages
500 http://sfo1.mirrors.digitalocean.com/mariadb/repo/10.3/ubuntu bionic/main arm64 Packages
1:10.3.20+maria~bionic 500
500 http://sfo1.mirrors.digitalocean.com/mariadb/repo/10.3/ubuntu bionic/main amd64 Packages
500 http://sfo1.mirrors.digitalocean.com/mariadb/repo/10.3/ubuntu bionic/main ppc64el Packages
500 http://sfo1.mirrors.digitalocean.com/mariadb/repo/10.3/ubuntu bionic/main arm64 Packages
1:10.1.44-0ubuntu0.18.04.1 500
500 http://cz.archive.ubuntu.com/ubuntu bionic-updates/universe amd64 Packages
500 http://cz.archive.ubuntu.com/ubuntu bionic-updates/universe i386 Packages
500 http://security.ubuntu.com/ubuntu bionic-security/universe amd64 Packages
500 http://security.ubuntu.com/ubuntu bionic-security/universe i386 Packages
1:10.1.29-6 500
500 http://cz.archive.ubuntu.com/ubuntu bionic/universe amd64 Packages
500 http://cz.archive.ubuntu.com/ubuntu bionic/universe i386 Packages

Naistalovat vybranou verzi aplikace pak lze následujícím příkazem.

sudo apt-get install mariadb-client=1:10.4.12+maria~bionic

Existuje ještě jeden snadnější způsob, který udělá přidání uložiště MariaDB za nás. V podstatě se jedná o externí skript, který lze spustit příkazem níže. [22]

curl -sS https://downloads.mariadb.com/MariaDB/mariadb_repo_setup | sudo bash

Po instalaci MariaDB serveru je doporučeno nastavit základní bezpečnostní prvky jako heslo uživatele a podobně. Konfiguraci v interaktivním módu lze provést spuštěním příkazu níže. [23]

sudo mysql_secure_installation

Po instalaci se automaticky spustí MariaDB server. To, že skutečně server běží je možno ověřit následujícím příkazem, kterým lze monitorovat stav DB serveru. [22]

systemctl status mysql

Po provedení příkazu se v ideálním případě zobrazí výpis podobný výpisu níže.

● *mariadb.service - MariaDB 10.4.12 database server*
Loaded: loaded (/lib/systemd/system/mariadb.service; enabled; vendor preset: enabled)
Drop-In: /etc/systemd/system/mariadb.service.d
└─migrated-from-my.cnf-settings.conf

Active: active (running) since Fri 2020-03-13 00:04:21 CET; 3 days ago
Docs: man:mysqld(8)
https://mariadb.com/kb/en/library/systemd/
Process: 1445 ExecStartPost=/etc/mysql/debian-start (code=exited, status=0/SUCCESS)
Process: 1443 ExecStartPost=/bin/sh -c systemctl unset-environment _WSREP_START_POSITION (code=exited, status=0/SUCCESS)
Process: 1135 ExecStartPre=/bin/sh -c [! -e /usr/bin/galera_recovery] && VAR= || VAR=`/usr/bin/galera_recovery`; [$? -eq 0] && systemctl set-environment _WSREP_START_POSITION=$VAR || exit
Process: 1133 ExecStartPre=/bin/sh -c systemctl unset-environment _WSREP_START_POSITION (code=exited, status=0/SUCCESS)
Process: 1127 ExecStartPre=/usr/bin/install -m 755 -o mysql -g root -d /var/run/mysqld (code=exited, status=0/SUCCESS)
Main PID: 1303 (mysqld)
Status: "Taking your SQL requests now..."
Tasks: 31 (limit: 4915)
CGroup: /system.slice/mariadb.service
└─1303 /usr/sbin/mysqld

MariaDB instalace není třeba v případě pouhé prezentace výsledů analýzy, neboť jsou k dispozici agregované výsledky ve formě parquet souborů, z kterých lze výsledky v prostředí aplikace Zeppelin rovněž zobrazit. Parquet soubor umožňuje ukládat tabulková data, a to včetně datových typů. Instalace MariaDB uložiště se tedy hodí pouze pokud chceme tyto parquet soubory za pomocí Sparku ze zdrojových dat vytvářet.

4.3 Java a Scala

Spark ve verzi 2.4.5. vyžaduje nainstalovanou Javu ve verzi 8. Ověřit si nainstalovanou verzi můžeme příkazem níže. [15]

java -version

V našem případě tento příkaz vypíše následující.

openjdk version "1.8.0_242"
OpenJDK Runtime Environment (build 1.8.0_242-8u242-b08-0ubuntu3~18.04-b08)
OpenJDK 64-Bit Server VM (build 25.242-b08, mixed mode)

Verze 8 je naprosto v pořádku a přesně to, co je pro náš případ potřeba. Může se nicméně stát, že Javu nainstalovanou nemáme, v takovém případě bychom nainstalovali Javu standartním příkazem níže.

apt install java

Takový příkaz ale nemusí naistalovat správnou verzi. V našem případě je nainstalována Java 11, s tou ale Spark není kompatibilní. Pro nainstalovaní Javy 8 je třeba použít příkaz níže.

sudo apt install openjdk-8-jdk

V případě již nainstalované Javy v odlišné verzi, než je požadovaná Java 8, instalace v systému zůstává souběžně s Javou 8. Poslední problém, který je třeba v tomto případě vyřešit, je zařídit, aby se defaultně používala právě Java 8. Naštěstí je toto v Linux prostředí jednoduché. Stačí nám následující příkaz.

sudo update-alternatives --config java

Po zadání příkazu se zobrazí všechny verze Javy, které jsou nainstalovány a k dispozici. Požadovanou verzi můžeme vybrat zadáním příslušného čísla dané verze. V našem případě je to verze číslo 2. Nabídka verzí je vypsána níže.

There are 2 choices for the alternative java (providing /usr/bin/java).

Selection	*Path*	*Priority*	*Status*
** 0*	*/usr/lib/jvm/java-11-openjdk-amd64/bin/java*	*1111*	*auto mode*
1	*/usr/lib/jvm/java-11-openjdk-amd64/bin/java*	*1111*	*manual mode*
2	*/usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java*	*1081*	*manual mode*

Press <enter> to keep the current choice[*], or type selection number:

Instalace Scaly je obdobná. Jelikož je použitá verze Spark frameworku zkompilována Scalou ve verzi 2.11, budeme potřebovat tuto verzi. Naštěstí je tato verze jediná, která je dostupná z uložiště Ubuntu v době psaní této práce. Stačí tedy příkaz níže. [24]

apt install scala

Obdobně jako verzi Javy můžeme ověřit verzi nainstalované Scaly příkazem níže.

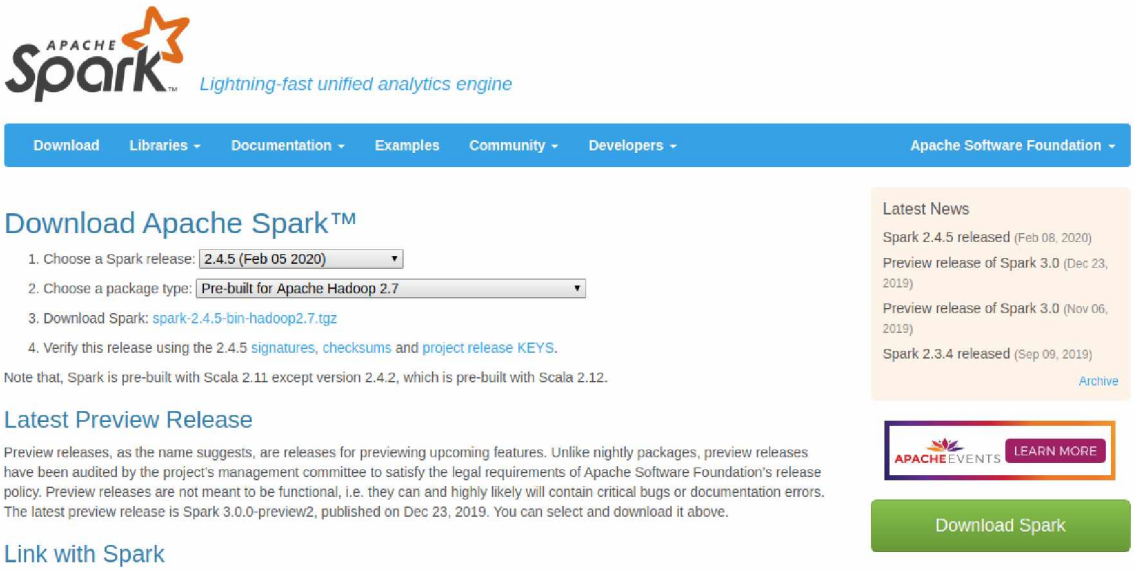
scala -version

Výpis předchozího příkazu by měl vypadat podobně jako je uvedeno níže.

Scala code runner version 2.11.12 -- Copyright 2002-2017, LAMP/EPFL

4.4 Spark

Spark je k dispozici zdarma ke stažení na stránkách výrobce na následujícím odkaze https://spark.apache.org/downloads.html. Na této stránce vybereme poslední stabilní verzi, která byla k dispozici v době zahájení práce na diplomové práci neboli verzi 2.4.5. Po vybraní požadované verze klikneme na stažení. Obrázek ze stránky výrobce, s vybranou poslední stabilní verzí je k dispozici níže. [24]



Obrázek 6 – Stahovaní Spark frameworku [24]

Jak již bylo řečeno, a také z obrázku je patrné, k vytvoření distribuce Sparku byla použita Scala ve verzi 2.11. Po stažení archivu se Sparkem je třeba extrahovat obsah do jakékoliv zvolené složky na disk v místě, které člověku provádějící instalaci dává smysl. Toto místo si je třeba zapamatovat, neboť dalším krokem je přidání lokace Sparku do proměnných operačního systému. Proměnné operačního systému lze zobrazit příkazem printenv. Naším cílem je do tohoto seznamu přidat proměnnou s názvem SPARK_HOME. Hodnota této proměnné je pak cesta do složky s obsahem staženého vyextrahovaného archivu se Spark instalací. Pro přidání systémové proměnné je třeba upravit soubor /etc/environment. To můžeme udělat v příkazové řádce třeba s použitím textového editoru Nano příkazem níže. [24]

sudo nano /etc/environment

Do tohoto souboru chceme uložit novou proměnou ve správném formátu a s korespondující hodnotou jako je to uvedeno v příkladu níže.

SPARK_HOME=/home/martin/installations/spark-2.4.5-bin-hadoop2.7.

Po uložení změn do souboru /etc/environment je třeba ještě následujícím příkazem informovat systém o změně v souboru, tak aby správně používal aktualizované hodnoty systémových proměnných. O to se postará příkaz níže.

source /etc/environment

Pro ověření můžeme znova zadat příkaz printenv, v následném výpisu už by měla být obsažena i nově vytvořená proměnná s názvem SPARK_HOME. Celkovou funkčnost Spark instalace

ověříme až později. Opět platí podobně jako u DB, že pro samotnou prezentaci výsledů není specifická verze Sparku potřeba a dobře poslouží i verze distribuovaná přímo s Zeppelinem.

4.5 Anaconda a Python

Instalaci Pythonu provedeme společně s instalací Anacondy. Distribuce Anacondy je ke stažení ze stránky výrobce na odkazu https://www.anaconda.com/products/individual. Náhled stránky je na následujícím obrázku.

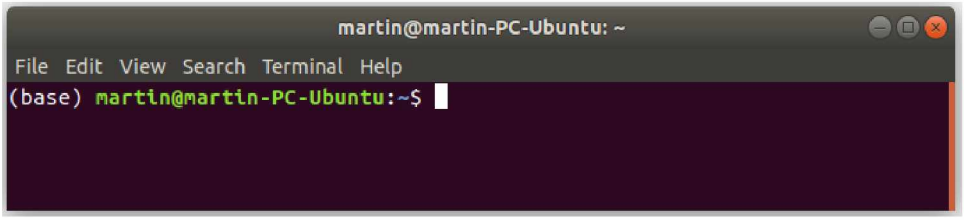
Obrázek 7 – Stažení Anaconda instalačního souboru [26]

Při stahování vybereme Python ve verzi 3.7. Po stažení spustíme instalační skript příkazem s cestou ke staženému souboru jako je tomu na ukázce níže.

bash ~/Downloads/Anaconda3-2019.10-Linux-x86_64.sh

Během instalace se skript dotáže na lokaci, kam chceme Anaconda distribuci nainstalovat. Abychom předešli zbytečným problémům, je dobré zvolit výchozí lokaci. Během instalace se skript dotáže také jestli chceme spustit příkaz "conda init". Tento příkaz zinicializuje defaultní "base" prostředí. Opět je doporučeno tento příkaz provést a nechat ho nastavit základní prostředí. [25]

Pro ověření instalace můžeme provést několik příkazů. Už při spuštění terminálu uvidíme nově před lokací v závorce prostředí. Pokud instalace proběhla korektně bude terminál vypadat jako na obrázku níže, ve kterém je aktivní "base" prostředí. [25]



Obrázek 8 – Ubuntu terminál s „base" prostředím

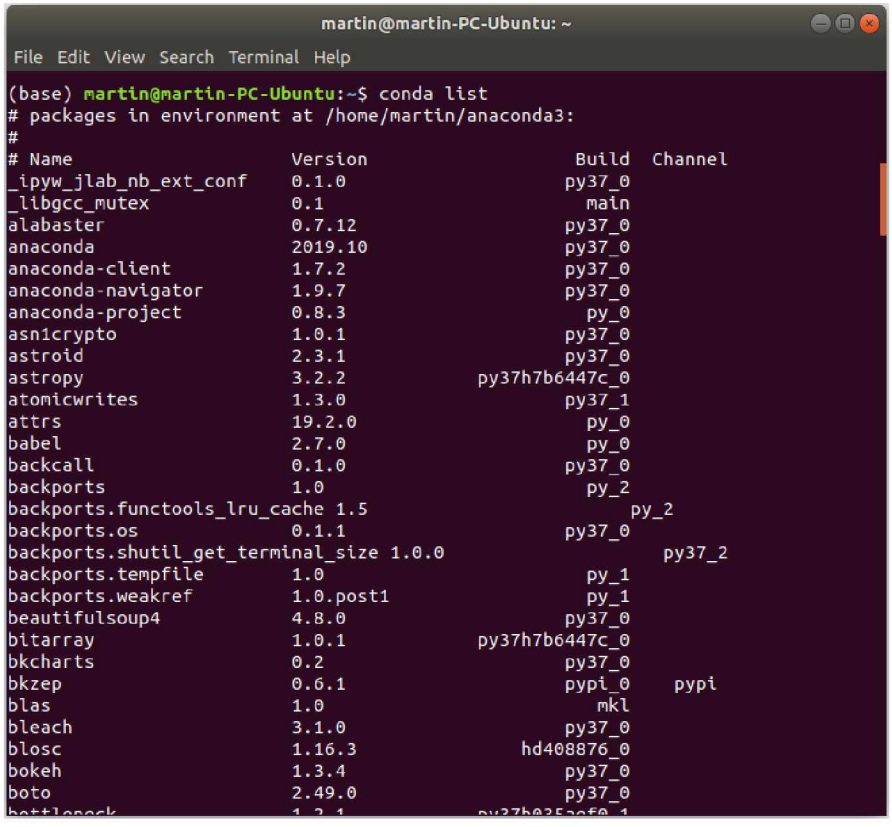
Instalaci Pythonu můžeme ověřit příkazem níže.

python –version

Po spuštění tohoto příkazu by se měla zobrazit verze nainstalovaného Pythonu jako v ukázce níže.

Python 3.7.4

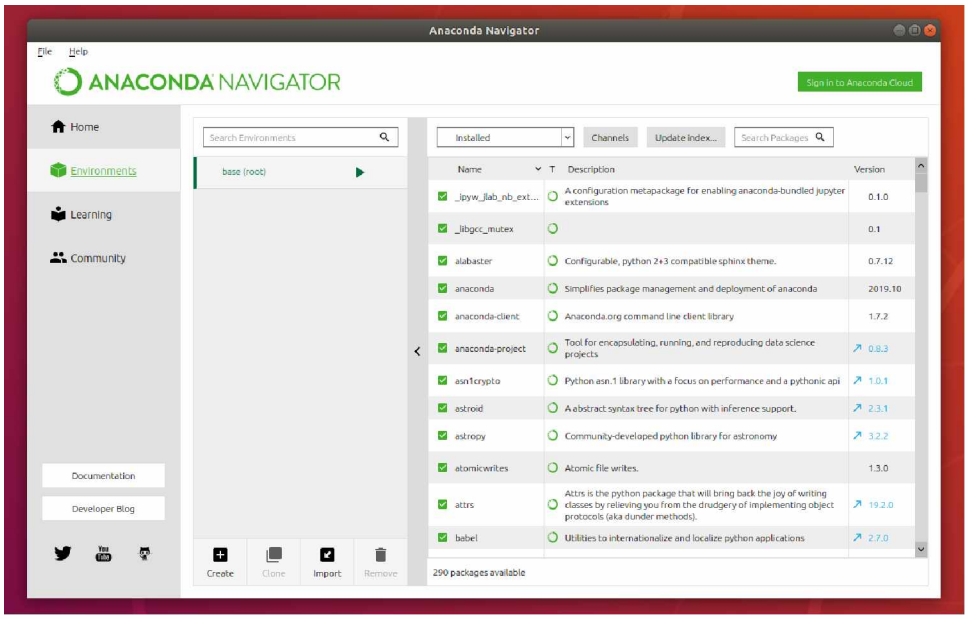
Další příkaz, který by měl fungovat je "conda list". Tento příkaz zobrazí nainstalované Anaconda balíčky a jejich verze podobně jako na obrázku níže. [25]



Obrázek 9 – Výpis nainstalovaných Python knihoven

Pozornému čtenáři neunikne, že některé balíčky mají vyplněny kanál. Ve většině případů se jedná o balíčky nainstalované jinou cestou než přes příkaz conda. Patrné je to třeba u bkzep balíčku, který nainstalujeme později přes balíčkovací systém Pip.

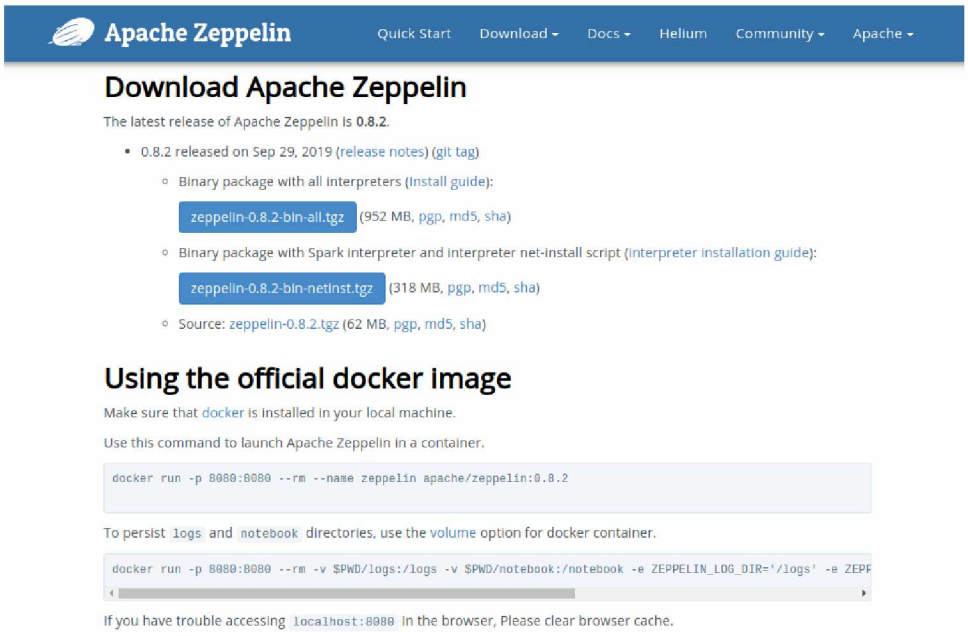
Poslední příkaz pro ověření instalace, je příkaz “anaconda-navigator”, který spustí nástroj s grafickým prostředím, jež umožňuje správu Anaconda balíčků v grafickém prostředí. Nástroj je na obrázku níže. [25]



Obrázek 10 – Ukázka okna aplikace Anaconda Navigator

4.6 Zeppelin server

Zeppelin server je možno stáhnout z odkazu https://zeppelin.apache.org/download.html. Webová stránka, na které je software k dispozici, je na obrázku níže.

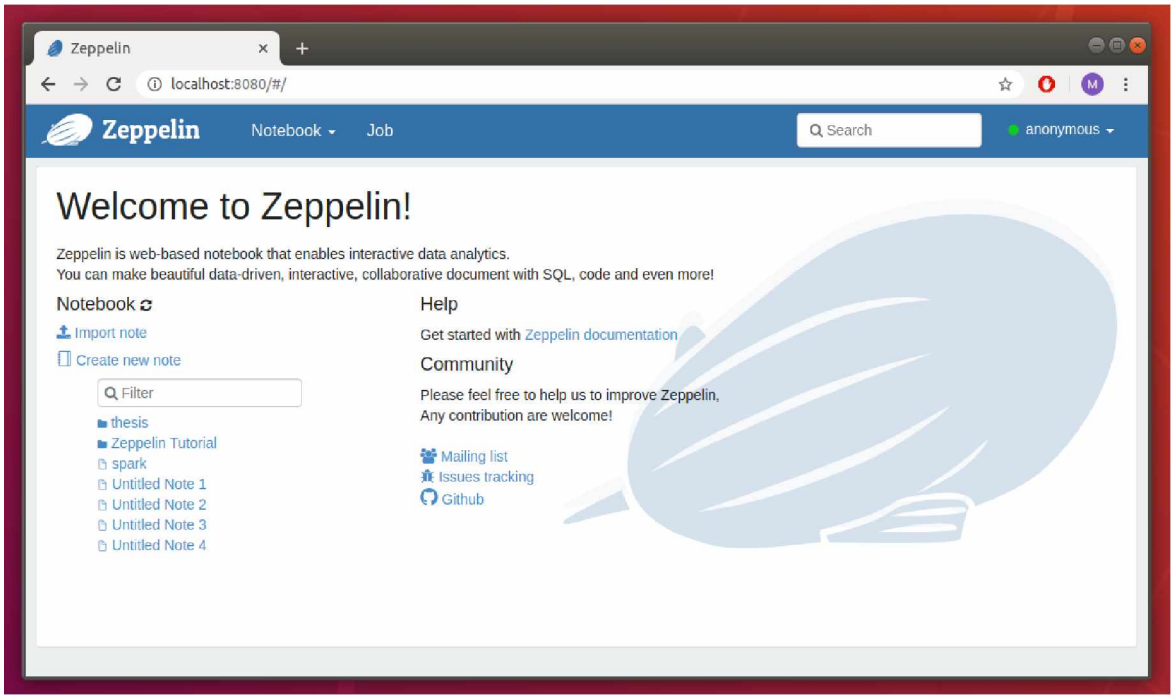


Obrázek 11 – Stažení aplikace Apache Zeppelin [10]

Po stažení souboru jeho obsah jednoduše extrahujeme do složky, kde chceme mít Zeppelin nainstalovaný, a to je ohledně instalace všechno. Instalaci můžeme ověřit tím, že v adresáři, kde máme naistalován Zeppelin přejdeme do složky bin. V této složce spustíme následující příkaz. [10]

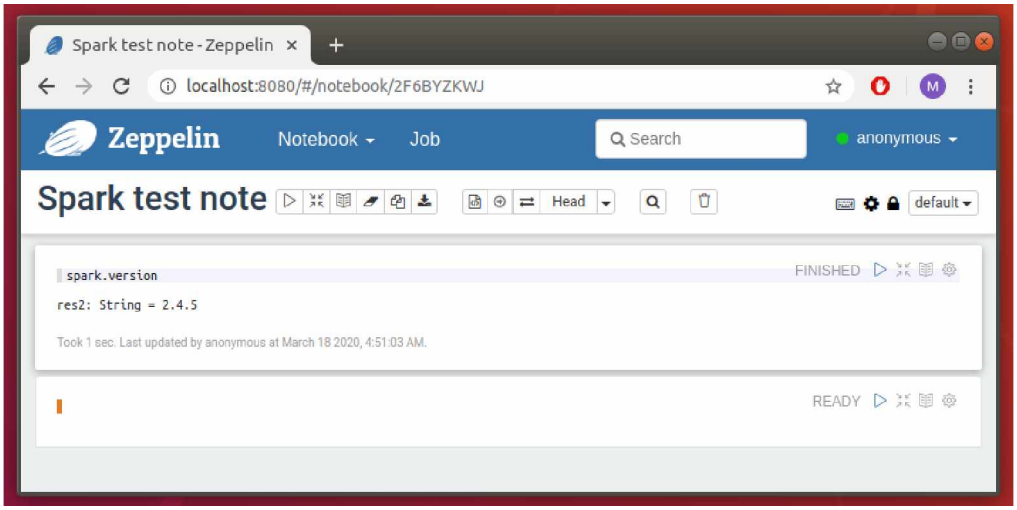
./zeppelin-daemon.sh start

Pokud vše funguje, jak má, spustí se Zeppelin webový server. Bez dalšího nastavení je třeba, aby byl volný port 8080, neboť tento port Zeppelin server využívá. Pokud se server spustil se stavem OK, je možné ve webovém prohlížeči zobrazit Zeppelin prostředí na adrese localhost:8080, ukázka tohoto prostředí je na obrázku níže. [10]



Obrázek 12 – Úvodní obrazovka aplikace Zeppelin

Ověřit, zda Zeppelin pracuje s naší lokální distribucí Spark frameworku můžeme snadno. Se správně nastavenou systémovou proměnnou SPARK_HOME nám při vytvoření nového sešitu s defaultním interpretem zvoleným jako "spark" následující příklad zobrazí verzi Sparku stejnou jako na následujícím obrázku.



Obrázek 13 – Zjištění používané Spark verze v aplikaci Zeppelin

Pokud je verze jiná, s nejvyšší pravděpodobností nižší, neboť Zeppelin verze 8.2.2 je distribuován společně s verzí Sparku 2.2.1. V tomto případě je třeba s největší pravděpodobností správně nastavit systémovou proměnou SPARK_HOME. [10]

Vypnutí Zeppelin serveru je možné obdobným způsobem jako spuštění. Příkaz je uveden níže.

./zeppelin-daemon.sh stop

4.7 Bokeh a Bkzep

Stažení a instalaci knihovny Bokeh provedeme příkazem níže.

conda install bokeh

Instalaci můžeme ověřit příkazem "conda list". V zobrazeném výpisu by balíček Bokeh měl být uveden. V době tvorby práce se nainstaloval balíček ve verzi 1.3.4, v průběhu tvorby práce sice vyšla tato knihovna ve verzi 2.0, ale pro naše účely zmíněná verze 1.3.4 svou funkcionalitou naprosto dostačuje.

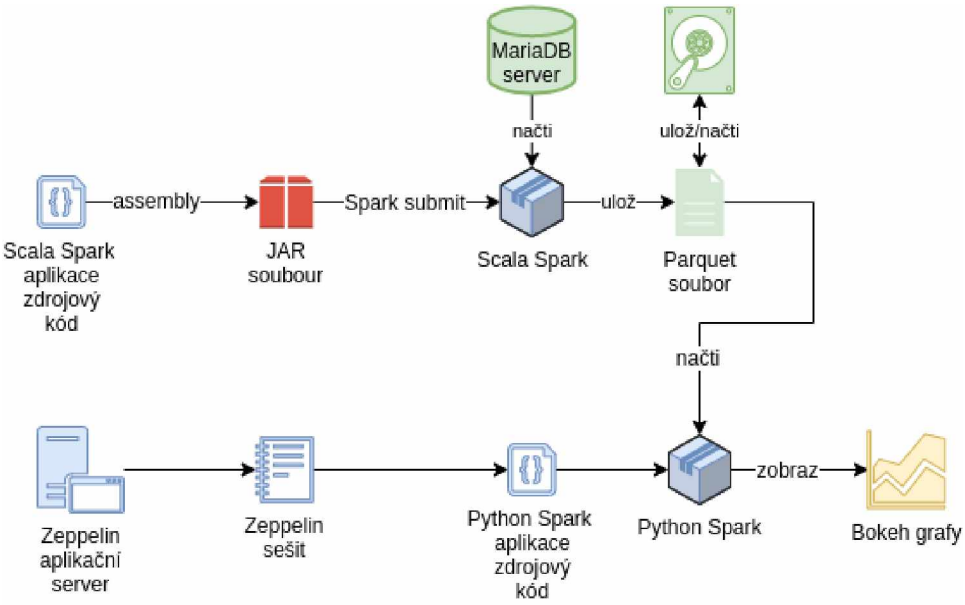
Jako poslední zbývá nainstalovat balíček Bkzep. Tento balíček není možné pomocí Condy nainstalovat, protože se na Anaconda uložišti nenachází, nicméně můžeme využít Python balíčkovací systém Pip. Příkaz pro instalaci je uveden níže.

pip install bkzep

Ověřit instalaci Bkzep lze stejným způsobem jako u knihovny Bokeh.

5. ANALÝZA VZOROVÝCH DAT

Pro praktický příklad byla zvolena data, která jsou výstupem algoritmu symbolické regrese. Jedná se o genetický algoritmus, který na vstupu obdrží soubor čísel. Výstupem algoritmu jsou rovnice, které popisují vstupní čísla. Pro ukončení algoritmu musí výstupní rovnice popisovat vstupní číslo se zadanou přesností, popřípadě je překročen počet cyklů, po které jsme rovnici ochotni hledat. Na obrázku níže je zobrazen diagram navrženého řešení od původního extrahování dat do zobrazení výsledků uživateli. [27]



Obrázek 14 – Navrhované řešení analýzy dat

V první části analyzování testových dat je potřeba nejprve vybrat vhodné údaje o experimentech, údaje, které mezi sebou budeme chtít porovnávat napříč jednotlivými experimenty a které by mohly nabízet zcela nový pohled na vypočtená data, popřípadě samotný průběh výpočtů.

Po pečlivém vybrání zajímavých dat ze vstupního souboru následuje další fáze. V této fázi je řešeno samotné získávání dat z jednotlivých archivů, které náleží jednotlivým experimentům. V ideálním případě nechceme obsah jednotlivých archivů extrahovat na disk ale rovnou v paměti načíst a uložit do databáze. Data se nehodí extrahovat jednoduše proto, že by zabírala zbytečné místo na disku, rovněž by to samozřejmě bylo více časově náročné, neboť archivy obsahují miliony a miliony drobných textových souborů a je zbytečné tím uložiště zatěžovat. Abychom se zbavili tohoto ohromného množství textových souborů je součástí této části také tvorba vhodného databázového schématu. Data z těchto malých souborů tedy jednoduše převedeme na vhodné databázové objekty a uložíme do nově vytvořeného databázového schématu.

Další fáze v analýze vzorových dat je zapojení Spark frameworku. V této fázi už jsou data z jednotlivých experimentů bezpečně uložena v korespondujícím databázovém schématu. Pro načtení dat z databáze a jejich následného zpracování je využito Scala Spark API. Výstupem této fáze budou parquet soubory. Alternativou pro parquet soubory by mohlo být uložit výstupy Scala Sparku do databáze. Jelikož je ale počítáno s více parquet soubory pro jednotlivé výstupy, mohlo by lehce dojít k znepřehlednění a pomotání výstupů. Další výhodou parquet souborů je jejich jednoduchý import a export. Parquet soubor lze lehce předat a načíst ho na úplně na jiném počítači. V případě využití integrované Spark databáze pro export bychom zůstali limitováni z hlediska opětovného načtení na konkrétní počítač, popřípadě cluster. Pro představení více přístupů a snadnější vývoj není v této fázi využito sešitové rozhraní Zeppelin, ale přímo vytvořený JAR soubor generovaný z vývojového prostředí IntelliJ IDEA. JAR soubor vytvoříme za pomocí SBT nástroje v kombinaci s SBT-Assembly pluginem, tak aby JAR soubor obsahoval všechny externí knihovny. Výsledný JAR soubor poté bude naservírován spark-submit skriptu který spustí samotnou Spark aplikaci a vykoná její instrukce. Těchto Scala Spark aplikací bude několik, konkrétně tři. Aplikace nicméně budou velice podobné, části pro načtení dat budou totožné, toto dělení je spíše kvůli přehlednosti, aby název aplikace vždy odpovídal tomu, co bude jejím výsledkem. Další výhodou samozřejmě bude menší doba, za kterou se samotné zpracování dat vykoná, neboť budou aplikace provádět méně úkonů.

Nevyužití Zeppelin sešitového rozhraní pro předchozí fázi má ještě jednu výhodu a tou je samotná práce s kódem. Zatímco vývojové prostředí nabízí nepřeberné množství funkcí pro usnadnění práce samotného programování, v Zeppelinu se nový kód především nováčkům s neznalostí Spark API píše podstatně hůře.

Po vytvoření jednotlivých parquet souborů s výsledky ze Scala Sparku bude využito sešitové rozhraní Zeppelin. Pro grafické zobrazení dat bude povolán na pomoc Python Spark, ten poslouží na opětovné načtení parquet souborů a poté bude využito toho, že pro Python existuje velké množství knihoven zabývající se vizualizacemi dat, včetně pro tento účel zvolené knihovny Bokeh, pomocí které budou vykresleny grafy, na které by integrované Zeppelin grafy nestačily. Jelikož lze s trochou dobré vůle napojit grafický výstup Bokeh vizualizací přímo do výstupu sešitů v Zeppelin rozhraní, měl by tento přístup nabízet uživatelsky přívětivý zážitek. Na jednom místě lze upravovat Python kód, z tohoto místa pak ten samý kód spustit a hned pod spuštěným kódem vidět samotnou vizualizaci. Uživatel může upravovat samotný kód nebo například vytvořit vstupní ovládací komponenty pro uživatele, jež nemají znalost Sparku nebo programovaní obecně, a i těmto uživatelům tak nabídnout více než jen statický graf s popisky.

5.1 Seznámení s daty

K dispozici máme několik skupin testů. Každý skupina obsahuje několik experimentů, tyto experimenty jsou pojmenovány podle generátorů náhodných čísel použitých při běhu genetického algoritmu. Každá skupina má své specifické jméno, které udává nějakou podstatnou informaci, kterou známe ohledně celé skupiny. Zjednodušeně lze říct, že je o experimentech známo, zda tyto testy například běžely ve více vláknech, nebo jestli tyto testy používají ten či onen datový typ, popřípadě zda se jedná o novou či starou verzi algoritmu. Skupiny, které jsou k dispozici, jsou k v seznamu níže.

- longdouble0
- longdouble5
- longdouble10
- nch
- nch_longdouble5
- singlethread_old
- singlethread_new
- multithread
- multithread_new

Obecně se jedná, jak již bylo řečeno, o výsledky algoritmu symbolické regrese, každá složka jednotlivých experimentů obsahuje rovněž zdrojové kódy experimentu, a to včetně pro analýzu důležitých souborů s nastavením konstant genetického algoritmu. Některé názvy souborů a jejich obsah jsou shrnuty v tabulce níže.

Název	Popis
Makefile	Soubor s nastavením sestavení pro jazyk C++.
ga.cpp; gpa.cpp	Části zdrojového kódu genetického algoritmu.
constantlist.cpp	Deklarace proměnných, do kterých se ukládají konstanty.
data_prototype.cpp	Jednotlivé kombinace vstupních hodnot, ke kterým genetický algoritmus hledá korespondující rovnice.
gpa_prototype.cpp	Inicializace generátorů náhodných čísel.
data_waves.cpp	Proměnné popisující vstupní data.
gpaprototype.cpp	Inicializace semínek generátorů náhodných čísel dle parametrů, s kterými byl spuštěn.

lego.cpp	Stavebnice, ze které se skládají stromy řešení.
my_rand.cpp	Soubor s jednotlivými generátory náhodných čísel.
experiment.h	Nastavení konstant genetického algoritmu.
gpaes0..n	Složky s výsledky genetického algoritmu. Každá složka obsahuje v názvu číslo, které bylo předáno jako semínko pro generátory náhodných čísel. V každé složce by se v ideálním případě měly nacházet tři soubory s výsledky, neboť vždy hledáme tři rovnice popisující tři proměnné na vstupu genetického algoritmu.
*.gpa	Textový soubor s výsledky genetického algoritmu.

Tabulka 3 – Soubory experimentů a jejich funkce

Po pečlivé analýze dat ze souborů, které jsou k dispozici, byly vybrány údaje z některých souborů, a jsou považovány za důležité. Zajímavé jsou například údaje ze souboru my_rand.cpp o použitých generátorech náhodných čísel. Ne vždy tento údaj ale koresponduje s názvem složky experimentu, tento název bude třeba rovněž uložit, neboť některé generátory mají v některých případech část semínek generátorů konstantní hodnotu jedna, zatímco ostatní semínka se v průběhu testu mění. Toto nastavení by mohlo být zajímavé pro následnou analýzu. Dále je důležitý soubor experiment.h, ve kterém se nacházejí konstanty genetického algoritmu. Lze tedy sledovat chování genetického algoritmu v závislosti na nastavení konstant napříč experimenty. Jako poslední jsou samotné výsledky genetického algoritmu v jednotlivých gpa souborech. Tyto soubory obsahují záznam o počtu vykonaných cyklů při běhu algoritmu, stejně tak obsahují dobu, po kterou algoritmus běžel a v neposlední řadě samotné řešení a jeho přesnost. Tento výsledek je zde uveden jako otisk všech členů v populaci seřazených podle ohodnocení řešení, kterému se říká v genetických algoritmech fitness. [27]

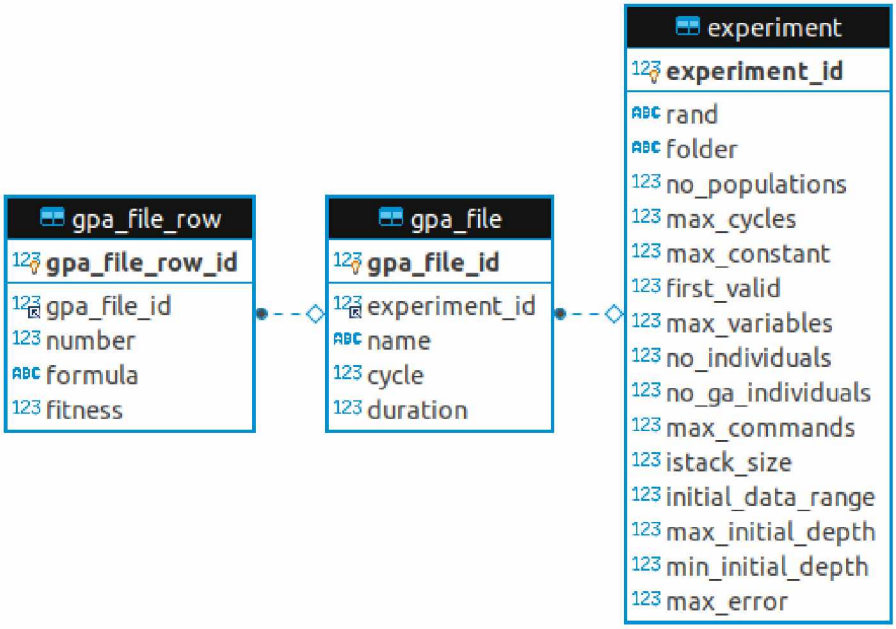
5.2 Databázové schéma pro vybraná data experimentů

Za účelem extrahování a uložení dat z jednotlivých experimentů je třeba vytvořit pro data vhodné databázové tabulky. Po předchozím zvolení pro analýzu potřebných dat získáme tři tabulky. První tabulka s názvem experiment popisuje jeden experiment z dané skupiny experimentů. K tomuto experimentu náleží jednotlivé gpa soubory. Jak již bylo zmíněno tyto gpa soubory jsou v našem případě v bezchybném a dokončeném běhu tři pro každou iteraci. Toto číslo odpovídá počtu hledaných nezávislých proměnných. Poslední tabulka reprezentuje jeden člen z populace, s jeho řešením a hodnotou ohodnocení fitness. Jednotlivé členy populace rovněž získáme z gpa souborů.

Následující blok ukazuje, jakými DDL příkazy lze vytvořit potřebné tabulky. Tyto příkazy lze aplikovat například v MariaDB interaktivní konzoli nebo v jakémkoli jiném více uživatelky přívětivějším databázovém správci. Pro potřeby diplomové práce byla vytvořena databáze s názvem thesis a DDL příkazy tedy pracují s touto DB.

```
CREATE TABLE thesis.`experiment`
(
    `experiment_id`      bigint(20) unsigned NOT NULL AUTO_INCREMENT,
    `rand`               varchar(30)  DEFAULT NULL,
    `folder`             varchar(100) DEFAULT NULL,
    `no_populations`     int(11)      DEFAULT NULL,
    `max_cycles`         int(11)      DEFAULT NULL,
    `max_constant`       int(11)      DEFAULT NULL,
    `first_valid`        int(11)      DEFAULT NULL,
    `max_variables`      int(11)      DEFAULT NULL,
    `no_individuals`     int(11)      DEFAULT NULL,
    `no_ga_individuals`  int(11)      DEFAULT NULL,
    `max_commands`       int(11)      DEFAULT NULL,
    `istack_size`        int(11)      DEFAULT NULL,
    `initial_data_range` int(11)      DEFAULT NULL,
    `max_initial_depth`  int(11)      DEFAULT NULL,
    `min_initial_depth`  int(11)      DEFAULT NULL,
    `max_error`          float        DEFAULT NULL,
    PRIMARY KEY (`experiment_id`)
) ENGINE = InnoDB
  AUTO_INCREMENT = 1
  DEFAULT CHARSET = utf8;

CREATE TABLE thesis.`gpa_file`
(
    `gpa_file_id`    bigint(20) unsigned NOT NULL AUTO_INCREMENT,
    `experiment_id` bigint(20) unsigned DEFAULT NULL,
    `name`           varchar(255)        DEFAULT NULL,
    `cycle`          int(11)             DEFAULT NULL,
    `duration`       float               DEFAULT NULL,
    PRIMARY KEY (`gpa_file_id`),
    KEY `gpa_file_experiment_fk` (`experiment_id`),
    CONSTRAINT `gpa_file_experiment_fk` FOREIGN KEY (`experiment_id`) REFERENCES
`ex-pe-riment` (`experiment_id`)
) ENGINE = InnoDB
  AUTO_INCREMENT = 1
  DEFAULT CHARSET = utf8;

CREATE TABLE thesis.`gpa_file_row`
(
    `gpa_file_row_id` bigint(20) NOT NULL AUTO_INCREMENT,
    `gpa_file_id`     bigint(20) unsigned DEFAULT NULL,
    `number`          int(11)             DEFAULT NULL,
    `formula`         varchar(2000)       DEFAULT NULL,
    `fitness`         float               DEFAULT NULL,
    PRIMARY KEY (`gpa_file_row_id`),
    KEY `gpa_file_row_gpa_file_fk` (`gpa_file_id`),
    CONSTRAINT `gpa_file_row_gpa_file_fk` FOREIGN KEY (`gpa_file_id`) REFERENCES
`gpa_file` (`gpa_file_id`)
) ENGINE = InnoDB
  AUTO_INCREMENT = 1
  DEFAULT CHARSET = utf8;
```

Z hlediska databázových technologií jsou zajímavé primární klíče, které využívají auto inkrementační funkci databáze MariaDB, což ulehčí práci, neboť se nemusíme starat o hodnotu primárního klíče u nově vytvářených záznamů. Dále si můžeme všimnout vazeb mezi jednotlivými tabulkami. Tyto vazby jsou realizovány referencemi na primární klíč z jiné tabulky. Vazby jsou patrné na následujícím obrázku, který reprezentuje všechny tabulky databáze thesis.



Obrázek 15 – Použité databázové schéma

Z obrázku je patrné, že se nejedná o nikterak složité schéma. Pro větší přehlednost o tom, odkud se jednotlivé hodnoty pro dané sloupce tabulek vyčítají je v následující tabulce tato informace k dispozici.

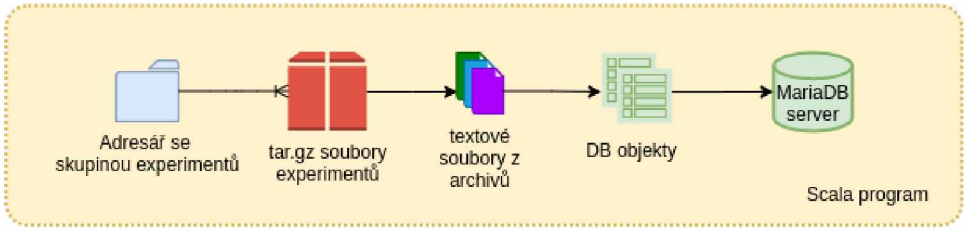
Tabulka	Sloupec	Zdroj
experiement	experiment_id	Primární klíč entity experiment, generován automaticky.
experiement	rand	Ze souboru my_rand.cpp vyčteno z prvního řádkového komentáře.
experiement	folder	Název složky sdružující experimenty stejné kategorie. Předán parametrem při spouštění nahrávacího programu.
experiement	no_populations	Konfigurační parametry genetického algoritmu ze souboru experiment.h.
experiement	max_cycles	

experiement	max_constant	Konfigurační parametry genetického algoritmu ze souboru experiment.h.
experiement	first_valid	
experiement	max_variables	
experiement	no_individuals	
experiement	no_ga_individuals	
experiement	max_commands	
experiement	istack_size	
experiement	initial_data_range	
experiement	max_initial_depth	
experiement	min_initial_depth	
experiement	max_error	
gpa_file	gpa_file_id	Primární klíč entity gpa_file, generován automaticky.
gpa_file	experiment_id	Reference na korespondující primární klíč z tabulky experiment, pod který daný gpa soubor patří.
gpa_file	name	Název gpa souboru, z kterého pochází hodnoty. Tímto názvem lze jednoznačně identifikovat a dohledat daný soubor.
gpa_file	cycle	Číslo udávající, ve kterém cyklu genetický algoritmus skončil vyčtené ze začátku gpa souboru.
gpa_file	duration	Číslo udávající, kolik sekund genetický algoritmus běžel vyčtené ze začátku gpa souboru.
gpa_file_row	gpa_file_row_id	Primární klíč entity gpa_file_row, generován automaticky.
gpa_file_row	number	Číslo označující pořadí člena v populaci. Členové jsou seřazeni od člena s nejmenší hodnotou fitness po největší. Platí, že čím menší ohodnocení fitness, tím vice se blížíme k řešení. Vyčteno z gpa souboru.
gpa_file_row	formula	Rovnice, jež je výsledkem genetického algoritmu. Vyčteno z gpa souboru.
gpa_file_row	fitness	Hodnota, jež hodnotí kvalitu řešení pro daného člena z populace, rovněž vyčteno z gpa souboru.

Tabulka 4 – Přehled jednotlivých tabulek a jejich sloupců

5.3 Program pro hromadné nahrání dat do databáze

Proces extrahování a uložení dat do jednotlivých tabulek databáze je třeba automatizovat. Pro tuto úlohu byl vybrán programovací jazyk Scala, a to zejména z důvodu jeho pozdějšího využití v kombinaci s frameworkem Spark, který je nativně napsaný ve Scale. Jedná se tedy o dobrou příležitost se s tímto jazykem blíže seznámit už v této fázi předzpracování dat. Na obrázku níže je zobrazen diagram toho, co výsledná Scala aplikace bude dělat.

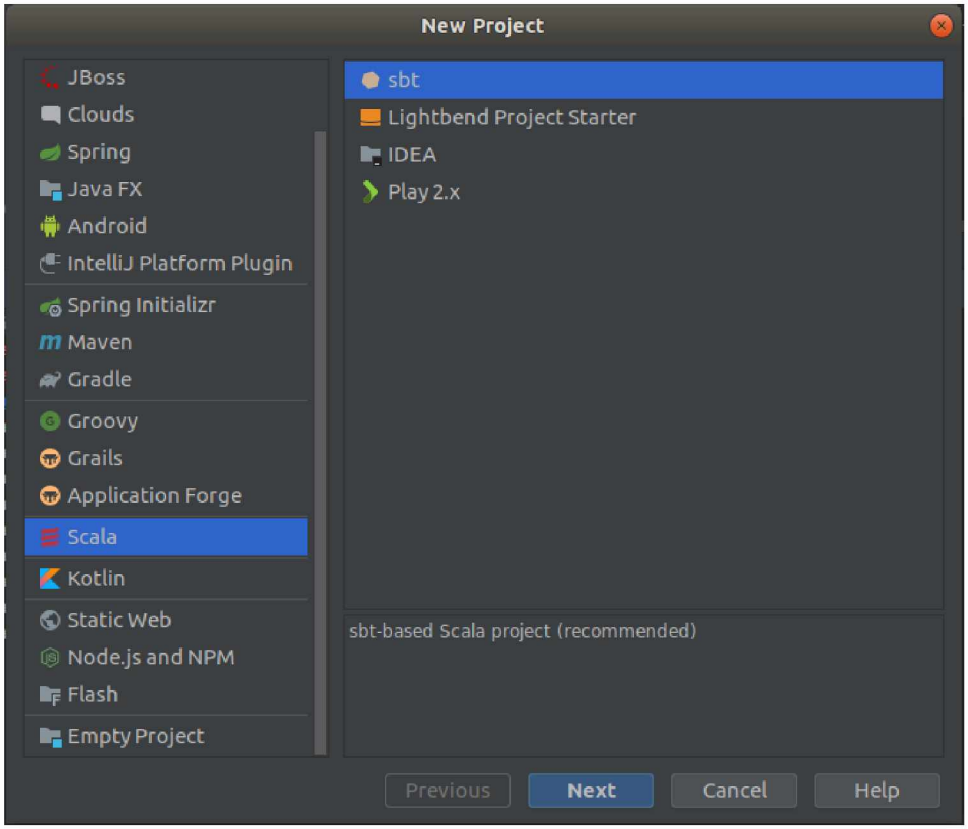


Obrázek 16 – Diagram aplikace pro extrahování a uložení dat experimentů

V zásadě se jedná o jednoduchou konzolovou aplikaci, ve které si není třeba hrát se složitým uživatelským rozhraním, neboť bude mít jen jednu úlohu a navržená aplikace bude využita jen jednou před začátkem analýzy dat. Konzolová aplikace na vstupu očekává jeden parametr a tím je cesta k adresáři ve kterém jsou jednotlivé tar.gz soubory provedených experimentů. Pro tvorbu této aplikace bude využito několika externích knihoven zejména pro přístup do databáze a zjednodušenou práci s databázovými entitami. Výsledný JAR soubor je výstupem SBT pluginu Assembly, který díky tomu obsahuje všechny zdrojové kódy externích knihoven. V následujících odstavcích bude vysvětleno několik konceptů použitých při tvorbě aplikace ve vývojem prostředí IntelliJ IDEA.

5.3.1 Nastavení projektu v IntelliJ IDEA

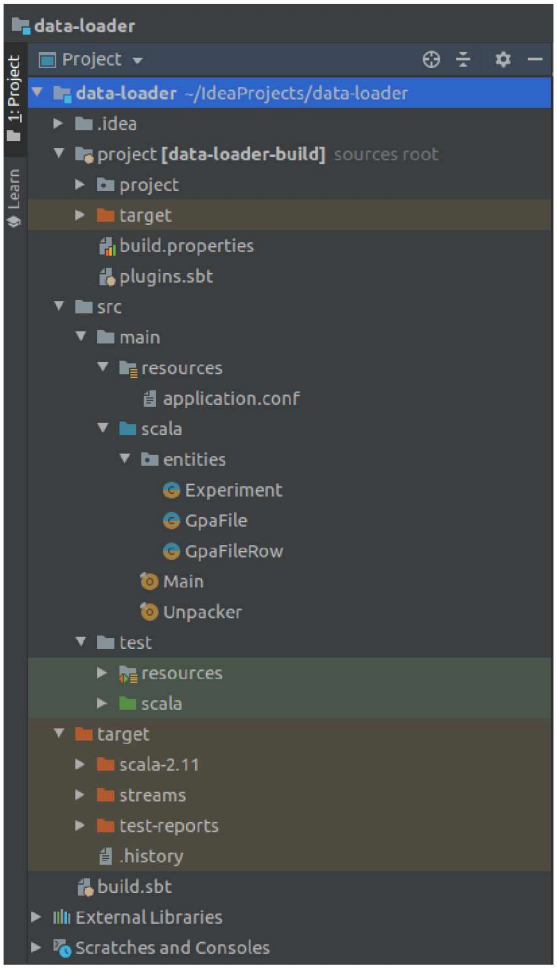
Abychom mohli pracovat s jazykem Scala ve vývojovém prostředí IntelliJ IDEA je třeba nainstalovat Scala plugin. Tento plugin přidává řadu výhod pro práci s jazykem Scala, namátkou lze zmínit asistenci při psaní kódu, a to včetně zvýraznění syntaxe, doplňovaní kódu, formátování a možnosti refaktorování. Další výhodou je integrace s SBT nástrojem sestavení, a to včetně integrované interaktivní SBT konzole. Samozřejmostí je pak Scala debugger a podpora psaní jednotkových testů například s technologií ScalaTest. Při vytváření nového projektu je třeba zvolit stejně jako na obrázku níže záložku Scala jazyk s nástrojem sestavení SBT.



Obrázek 17 – Vytvoření nového projektu SBT Scala projektu

5.3.2 Struktura SBT projektu

Popis samotné tvorby aplikace zde není popsán, nyní se tedy můžeme přesunout do stavu, kdy už je program kompletní a následovat bude popis některých klíčových vlastností programu. Jako první nás bude zajímat struktura projektu jež je nazván jednoduše data-loader neboli nahrávač dat. Zmíněná struktura je patrná na obrázku níže.



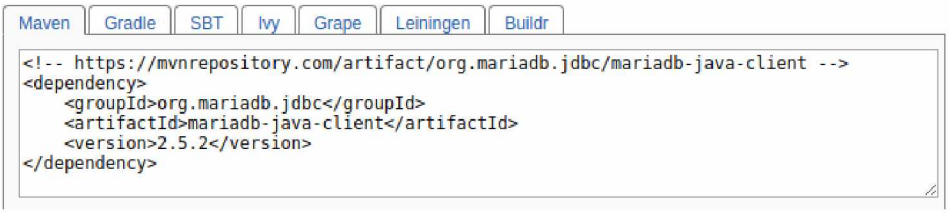
Obrázek 18 – Struktura aplikace pro nahrání dat do DB

Z obrázku vyplývá, že se jedná o projekt velice jednoduchý. Projekt má standartní SBT strukturu, která se podobá struktuře Maven Java modulů. Ve složce se zdrojovými kódem si lze všimnout v adresáři /src/main/resources konfiguračního souboru application.conf. Tento soubor obsahuje konfigurační parametry spojené s databázovým připojením, které bude více popsáno v další podkapitole. V adresáři /src/main/scala jsou jednotlivé soubory zdrojového kódu. Nechybí zde objekt Main obsahující metodu main, která slouží jako vstupní bod do aplikace. Dále je zde definován objekt Unpacker, jež složí k procházení a zpracovávání dat uvnitř tar.gz archivů. Je to místo s téměř veškerou potřebnou logikou pro zpracování dat. V balíčku entities se nacházejí třídy reprezentující databázové entity pro ORM, jehož koncept je vysvětlen dále v této kapitole. Adresář /src/test je místo pro jednotkové testy, které pro tuto chvíli v aplikaci nejsou. Adresář /target je místo do kterého se ukládají soubory po sestavení aplikace. V kořenu projektu se dále nachází soubor build.sbt, tento soubor je základem každé SBT aplikace. V ukázce níže je obsah build.sbt souboru aplikace na nahrávání dat.

```
name := "data-loader"
version := "0.1"
```

```
scalaVersion := "2.11.12"

libraryDependencies += "org.mariadb.jdbc" % "mariadb-java-client" % "2.5.2"
libraryDependencies += "org.scalikejdbc" %% "scalikejdbc" % "3.4.0"
libraryDependencies += "org.scalikejdbc" %% "scalikejdbc-config" % "3.4.0"
libraryDependencies += "org.skinny-framework" %% "skinny-orm" % "3.0.3"
libraryDependencies += "org.apache.commons" % "commons-compress" % "1.20"
```

V souboru zobrazeném v ukázce se nachází nastavení projektu, definují se zde externí knihovny a mnoho dalšího. Z obsahu je patrné, že došlo k použití celkem pěti externích knihoven. První čtyři knihovny usnadňují práci s databází a připojením do databáze, poslední pak slouží k manipulaci s archivy. SBT konfigurace se opět hodně podobá Mavenu, dokonce základní uložiště, které SBT využívá pro hledání externích knihoven je stejné jako u Mavenu, tedy Maven Central Repository dostupné na stránce https://mvnrepository.com/repos/central. Toto uložiště nabízí knihovny pro Scalu i pro Javu, a jelikož Scala umožňuje používání Java knihoven je tato komptabilita Maven uložišť s SBT velice výhodná. S Mavenem SBT sdílí i terminologii. Pro ukázku můžeme na následujícím obrázku vidět zápis přidání definice externí knihovny pro Maven z centrálního Maven uložiště.



Obrázek 19 – Ukázka Maven závislosti pro knihovnu MariaDB JDBC klienta [29]

Stejnou logikou tedy přidáváme závislosti na externí knihovny v rámci SBT. Zápis je nicméně zkrácený stejně jako v ukázce projektového build.sbt souboru. Mezi groupId a artifatcId je jeden nebo dva znaky reprezentující procento. Dva v případě, že se jedná o knihovnu v jazyce Scala, jeden v případě Java knihovny jako je například MariaDB klient. Pokud chceme přidat knihovnu pro Scalu, musí se daná knihovna na uložišti nacházet v kompatibilní verzi se Scalou daného projektu. Na obrázku níže je vidět příklad pro Scala externí knihovnu ScalikeJDBC, která je ve verzi 3.4.1 k dispozici pro Scalu ve verzi 2.11, 2.12 a 2.13.

Obrázek 20 – Seznam verzí knihovny ScalalikeJDBC pro různé verze jazyka Scala [30]

Jako poslední je třeba zmínit soubor v adresáři /project a to plugin.sbt, který slouží k definování pluginů pro SBT. V tomto souboru je definice pro přidání již zmíněného pluginu SBT-Assembly. Definice je k dispozici níže.

```
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.14.10")
```

5.3.3 Nastavení připojení do databáze

Pro připojení do MariaDB potřebujeme v jazyce Scala kompatibilní JDBC ovladač. Tento ovladač se jmenuje MariaDB Java Client, a je k dispozici ke stažení na centrálním Maven uložišti. Pro usnadnění práce s JDBC API je využita knihovna s názvem ScalikeJDBC. Tato knihovna obaluje původní JDBC API a nabízí přehlednější a lepší práci s databází, dotazy a výsledky, které dotazy vrací. Knihovna ScalikeJDBC je použita s konfiguračním modulem umožňující snadné nastavení a inicializování potřebných objektů pro práci s DB. Přihlašovací údaje k DB jsou obsahem souboru /src/main/resources/application.conf, obsah tohoto souboru je v ukázce níže.

```
# JDBC settings
db.default.driver="org.mariadb.jdbc.Driver"
db.default.url="jdbc:mariadb://localhost:3306/thesis"
db.default.user="root"
db.default.password="password"
# Connection Pool settings
db.default.poolInitialSize=10
db.default.poolMaxSize=20
db.default.connectionTimeoutMillis=1000
```

Dochází zde k propojení MariaDB JDBC ovladače s knihovnou ScalikeJDBC a to pomocí

konfigurační položky db.default.driver. Dále je zde nastavena adresa serveru a port na kterém služba běží, jméno databáze, uživatele a jeho heslo, a v neposlední řadě také nastavení skupiny připojení a jejich parametry jako maximální počet souběžných připojení, doba, po které se připojení ukončí při ztraceném spojení a podobně. ScalikeJDBC podporuje více implementací tříd které se starají právě o vytváření a udržování více připojení do DB. API, které je třeba implementovat, aby ho knihovna ScalikeJDBC podporovala se nazývá Database Connection Pooling, v základu ScalikeJDBC využívá implementaci s názvem Apache Commons DBCP. ScalikeJDBC parametry lze nastavit i přímo v kódu, tento přístup je však méně přehledný, další nevýhoda je nutnost nového sestavení, pokud by se například změnily přístupové údaje do DB, z tohoto důvodu byl zvolen konfigurační soubor.

Tento soubor lze tedy přepsat ve výsledném JAR souboru bez nutnosti znova kompilovat program. Je pro to potřeba jen zvolit vhodný nástroj kterým lze otevřít JAR soubor. Pokud ovšem bude uživatel mít databázového uživatele root s heslem password a vytvořenou databázi thesis, nemusí už do archivu přistupovat vůbec.

ScalikeJDBC se umí připojit do různých relačních databází, namátkou například PostgreSQL, MySQL, MariaDB nebo třeba databáze pracující pouze v operační pamětí H2. Díky tomu tato knihovna nabízí flexibilitu při přechodu na jiné databázové řešení nebo v případě H2 usnadňuje testování a vývoj samotné aplikace. Dále je možné používat více databází najednou, a to jak různých, tak stejných typů. Databázi lze v konfiguračním souboru pojmenovat a podle tohoto jména pracovat vždy jen s DB s kterou je potřeba v daný okamžik pracovat.

Díky konfiguračnímu modulu lze získat připojení do DB velice snadno. Ukázka níže otevře nastavený počet připojení pro veškeré databáze v konfiguračním souboru a vytvoří objekt typu AutoSession, ve zkratce jde o výchozí hodnotu pro všechny parametry typu DBSession označené klíčovým slovem implicit. Díky tomu nemusíme tento parametr předávat do SQL bloků.

```
DBs.setupAll()
implicit val session = AutoSession
```

Toto nastavení je vhodné použít na začátku aplikace, a naopak na konci po sobě uklidit a zavřít všechna spojení, jako je to v ukázce níže.

```
DBs.closeAll()
```

5.3.4 Procházení archivů a práce s textovými soubory v jazyce Scala

Poté co bylo úspěšně dosaženo připojení k DB můžeme přemýšlet o tom, jak s databází pracovat efektivně a snadno. O databázovém schématu už bylo rozhodnuto a potřebné tabulky byly DDL skriptem vytvořeny. Abychom mohli začít ukládat data do databáze je třeba tyto data vyčíst z textových souborů uvnitř tar.gz archivů. K tomuto účelu byla využita knihovna Apache Commons Compress.

Abychom mohli s TGZ soubory pracovat je dobré jim nejdříve rozumět. Tar soubor je kolekce sdružující více souborů do jednoho pro přehlednější ukládání. Tar soubory jsou často komprimovány z důvodu snížení jejich velikosti, to je okamžik kdy tar soubor dostane druhou koncovku gz označující komprimovaný soubor. Výsledná aplikace musí umět procházet TGZ soubory a vyčítat z jednotlivých textových souborů potřebné hodnoty, tyto hodnoty následně uložit do korespondujících databázových tabulek. Z důvodu šetření místa na pevném disku nebudeme všechny TGZ soubory extrahovat na disk, ale rovnou ukládat informace přímo do DB. Ukázka níže zobrazuje blok zdrojového kódu, konkrétně jednu funkci, která z argumentu předaného při spuštění programu nalezne všechny soubory s příponami předané v kolekci list.

```
def getListOfFiles(dir: String, extensions: List[String]): List[File] = {
  val d = new File(dir)
  if (d.exists && d.isDirectory) {
    d.listFiles.filter(_.isFile).toList.filter { file =>
      extensions.exists(file.getName.endsWith(_))
    }
  } else {
    List[File]()
  }
}
```

Ukázka nejdříve z předané cesty vytvoří objekt typu File. Poté dojde k ověření, zda se jedná o složku, pokud ano, dojde k filtrovaní všech souboru ve složce podle předaných přípon.

Poté co je k dispozici list všech TGZ souborů je třeba začít se samotným zpracováním. Kolekci souborů typu File transformujeme na kolekci typu FileInputStream, a každý z těchto input streamů předáme na zpracování objektu Unpacker. Unpacker nejprve provede dekompresi za pomocí CompressorInputStream objektu z knihovny Apache Commons Comress. Díky tomuto streamu můžeme číst jednotlivé soubory uvnitř komprimovaného souboru. Tento stream sám o sobě nicméně nestačí, musíme z něho udělat jiný stream a to ArchiveInputStream. ArchiveInputStream nabízí potřebné API pro čtení souborů uvnitř tar souboru. Za pomocí metody getNextEntry můžeme přistupovat už k samotným souborům s výsledky experimentů jako jsou

experimetn.h nebo třeba my_rand.cpp. V další ukázce zdrojového kódu je k nahlédnutí blok starající se právě o traverzování napříč jednotlivými soubory uvnitř tar archivu.

```
var latestEntry: ArchiveEntry = archiveInputStream.getNextEntry
var gpaFiles: mutable.MutableList[(ArchiveEntry, Iterator[String])] =
mutable.MutableList[(ArchiveEntry, Iterator[String])]()
var experimentId: java.lang.Long = null
var randCode: String = null

while (latestEntry != null) {
  if (!latestEntry.isDirectory) {
    if (List("my_rand.cpp").exists(latestEntry.getName.endsWith(_))) {
      randCode = processRandFile(archiveInputStream)
    }
    if (List("experiment.h").exists(latestEntry.getName.endsWith(_))) {
      experimentId = processExperimentFile(archiveInputStream, dirName)
    }
    else if (List("gpa").exists(latestEntry.getName.endsWith(_))) {
      gpaFiles += (latestEntry -> Source.fromInputStream(archiveInputStream)
        .getLines().toList.toIterator)
    }
  }

  latestEntry = archiveInputStream.getNextEntry
}
```

Po získání jednotlivých textových souborů z TGZ archivů je třeba s nimi umět pracovat. Soubory je nejprve zapotřebí převést z ArchiveInputStream typu do textového datového typu. Postup je v ukázce kódu níže.

```
val experimentFileAsString = Source.fromInputStream(is).mkString
```

V tomto bodě už máme každý soubor k dispozici v textovém datovém typu. Tento přístup je využit pro soubory experiment.h a my_rand.cpp. Pro analýzu těchto souborů využijeme regulárních výrazů a je nám jedno na jakém řádku se co v daném souboru nachází. Nicméně v případě gpa souborů chceme mít možnost pracovat s textovým souborem po jednotlivých řádcích reprezentující jednotlivé členy v populaci genetického algoritmu. V jazyce Scala lze získat jednotlivé řádky elegantním způsobem jako je to ukázáno na ukázce níže.

```
gpaFiles += (latestEntry -> Source.fromInputStream(archiveInputStream)
  .getLines().toList.toIterator)
```

Takto lze jednotlivé řádky za pomoci získaného iterátoru snadno procházet. Následující ukázka kódu prezentuje využití regulárního výrazu k získání hodnoty konstanty ze souboru experiment.h.

```
def findConstantValue(experimentFileText: String, constantName: String): String =
{
```

```
  val keyValPattern: Regex = s"""\\b$constantName\\b\\s([0-9.]+)""".r
  for (patternMatch <- keyValPattern.findFirstMatchIn(experimentFileText)) {
    return patternMatch.group(1)
  }
  null
}
```

Regulární výrazy umožňují najít definovanou část textu odpovídající popisu. Definovat lze v podstatě cokoliv od specifické kombinace písmen po různé speciální znaky nebo třeba konec řádku. Vytvoření vzorce, jež definuje regulární výraz je ve Scale jednoduché, k dispozici je pro tento účel metoda r na datovém typu String. Metoda r vrátí instanci objektu Regex, s pomocí kterého lze zavolat hledání řetězce v předaném textu. V ukázce si je možné povšimnout jednoduchosti s jakou lze ve Scale využít String interpolaci, a tím v textovém řetězci nahradit constantName za hodnotu proměnné s využitím znaku $. Další funkce regulárních výrazů je možnost získat část výsledku ohraničenou závorkami. Takovou část výsledku poté můžeme vyčíst jako v ukázce z výsledného objektu po hledání metodou group, do které jako parametr předáme číslo skupiny podle toho, kolikátou skupinu z daného regulárního výrazu chceme získat. Číslováno je od jedničky a pořadí odpovídá prvnímu výskytu skupiny ve vzorci regulárního výrazu. Pro přehlednost náš vzorový regulární výraz pojmenovaný regex říká najdi přesnou kombinaci znaků předanou proměnou constantName následovanou mezerou, která je následovaná zachycovanou skupinou. Tato skupina může obsahovat libovolný počet znaků jež mohou být pouze čísla nebo tečka reprezentující desetinnou čárku. Po vykonání hledání s tímto vzorcem bude ve skupině jedna, za předpokladu nalezení takového vzorce v textu, hodnota dané konstanty.

5.3.5 ORM a samotné ukládání dat do DB

Před samotným ukládáním do databáze pomocí DML příkazů si můžeme práci s DB zjednodušit pomocí ORM neboli objektově relačního mapování. Tento koncept se stará o konverzi dat z DB do objektů programovacího jazyka a naopak. Zjednodušeně můžeme například vytvořit objekt který bude odpovídat DB entitě a tento objekt provoláním nějaké pomocné generické metody uložit do DB. Naopak lze získávat výsledky dotazů přímo jako objekt, popřípadě kolekci více stejných objektů. ORM dále přináší například výhodu implicitní kontroly datových typů pro jednotlivé objekty.

Pro získání funkcionality ORM je v projektu použita knihovna Skinny-ORM. V následující ukázce je prezentováno, jakým způsobem lze vytvořit objekt reprezentující DB entitu gpa_file.

```
case class GpaFile(gpa_file_id: Long, experiment_id: Long, name: Option[String],
cycle: Option[Int], duration: Option[Float])

object GpaFile extends SkinnyCRUDMapper[GpaFile] {
```

```
  override lazy val defaultAlias = createAlias("g")
  override lazy val primaryKeyFieldName = "gpa_file_id"

  override def extract(rs: WrappedResultSet, n: ResultName[GpaFile]): GpaFile =
new GpaFile(
    gpa_file_id = rs.get(n.gpa_file_id),
    experiment_id = rs.get(n.experiment_id),
    name = rs.get(n.name),
    cycle = rs.get(n.cycle),
    duration = rs.get(n.duration))
}
```

Je patrné, že byla vytvořena case class s názvem GpaFile. Tato třída odpovídá dle parametrů databázové tabulce gpa_file, a lze do ní načíst jeden řádek této tabulky. Pro zpřístupnění obecných metod je třeba podědit z objektu SkinnyCRUDMapper, kde je jako generický typ použita právě case class GpaFile. Tímto si zpřístupníme metody pro DML příkazy jako insert, update a delete společně se základní ORM funkcí samotného mapování. V následující ukázce je zobrazeno uložení jednoho záznamu do tabulky gpa_file.

```
val gpaFileId: Long = GpaFile.createWithAttributes('experiment_id -> experimentId,
'name -> archiveEntry.getName, 'cycle -> cycle.toInt, 'duration ->
duration.toFloat)
```

Takto jednoduše lze skutečně uložit záznam do DB. Po úspěšném vykonání vložení pomocí metody createWithAttributes získáme automaticky vygenerovaný primární klíč, který lze později využít při ukládání záznamů do tabulky gpa_file_row. V další ukázce je příklad DML operace update entity experimet. Pro porovnání jsou k dispozici dva způsoby, první bez funkcionality objektu SkinnyCRUDMapper a druhý s pomocí funkcionality tohoto objektu, ve výsledku ale oba způsoby dělají to samé.

```
// 1. zpusob
DB localTx { implicit session =>
  withSQL {
    update(Experiment).set(Experiment.column.rand -> randCode).
      where.eq(Experiment.column.experiment_id, experimentId)
  }.update.apply()
}

// 2. zpusob
Experiment.updateById(experimentId).withAttributes('rand -> randCode)
```

Z ukázky je patrné, že druhý způsob umožňuje více zkrácený zápis. První ukázka ukazuje způsob, jak by update mohl vypadat bez použití knihovny Skinny-ORM, a ukazuje tak jednu z mnoha výhod Skinny-ORM knihovny.

V průběhu práce na programu pro nahrávání dat se ukázalo, že velké množství insert příkazů vykonávaných za sebou je značně pomalé. Autor toto chování přisuzoval režii, která vzniká s každou transakcí při jednotlivých DML příkazech pro vložení. Řešením tohoto neduhu by mohl být hromadný příkaz na vložení. Hromadné vložení zajistí vložení více záznamu v rámci jedné databázové transakce. Bez většího zásahu lze hromadné vkládání využít pro vložení všech záznamů do tabulky gpa_file_row, vždy pro jednu entitu gpa_file. ScalikeJDBC nabízí podporu pro hromadné vkládání, samotný kód tohoto procesu je k dispozici v ukázce níže.

```
var params: Seq[Seq[Any]] = Seq();
  while (lines.hasNext) {
    val experimentRowStringPieces = lines.next().split(":", 2)
    val number = experimentRowStringPieces.head.replaceAll("\\s+", "").toInt
    val restOfLine = experimentRowStringPieces.last.split(" {3}").filter(s =>
!s.isEmpty)
    val formula = restOfLine.head.trim
    val fitness = restOfLine.last.split('=').last.toFloat
    params = params :+ Seq(gpaFileId, number, formula, fitness)
  }

  DB localTx { implicit session =>
    sql"insert into gpa_file_row (gpa_file_id, number, formula, fitness) values
(?, ?, ?, ?)".batch(params: _*).apply()
  }
}
```

Ve zkratce v ukázce dochází ke čtení jednotlivých řádků gpa souboru. Hodnoty jednotlivých členů populace jsou ukládány do proměnné s názvem params. Hodnoty pro všechny členy populace jsou poté předány metodou batch do SQL DML příkazu. Tyto hodnoty nahrazují v příkazu pro vložení jednotlivé otazníky. Zajímavý je v ukázce operátor *_, říkající kompilátoru, aby předal každý prvek ze sekvence samotně, a ne celou sekvenci najednou. V praxi to znamená že metoda batch byla zavolána tolikrát, kolik má sekvence členů.

Zbývá ověřit to nejpodstatnější, jestli a o kolik se nahrávání do databáze zrychlilo. Ověření bylo vykonáno na reálných datech ze skupiny experimentů nch_longdouble5, konkrétně na archivu s názvem DDDDDddd.tar.gz. Doba, po kterou vkládání dat z tohoto archivu trvalo je v tabulce níže.

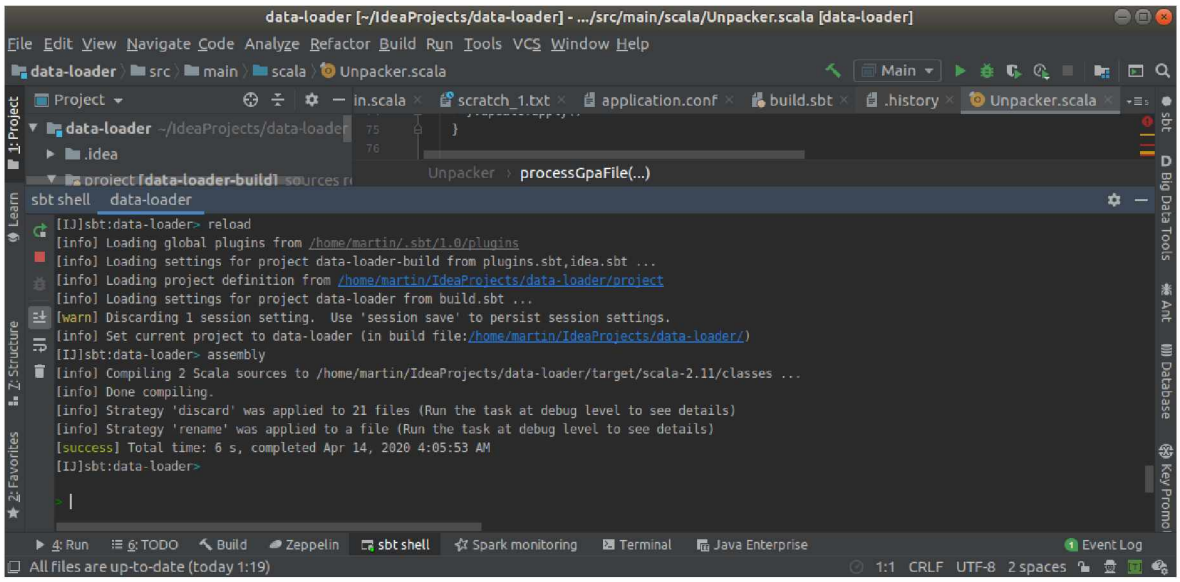
Metoda	Čas zpracování celého archivu [ns]
Jednotlivé vkládání do gpa_file_row	14180243947286
Hromadné vkládání do gpa_file_row	758459477249

Tabulka 5 – Porovnání vkládání záznamů do DB jednotlivě a hromadně

Z naměřených hodnot vyplývá, že došlo k významnému zrychlení procesu nahrávání dat. Ukládání dat je nyní skoro devatenáctkrát rychlejší. Pro další zrychlení by bylo třeba více upravit kód, tak například hromadně nahrával i tabulku gpa_file. Toto by znamenalo vytvoření nějaké vhodnější datové struktury pro dočasné ukládání tak, aby jednotlivé záznamy gpa_file_row tabulky byly přiřazeny ke správné entitě typu gpa_file. Dále by se určitě dalo zaměřit na počet záznamů nahraných v jedné transakci, a vybrat tak optimální počet pro hromadné vkládání. V našem případě je toto číslo dané počtem členů v rámci jednoho gpa souboru, ale s dodatečnými úpravami v kódu, by toto mohlo jít nastavit parametricky. Jako hlavní důvod, proč rychlost nahrávání dál v tomto bodě neoptimalizovat je fakt, že tento program je použit jednorázově pro prvotní zpracování a uložení dat a poté se již znova nepoužije. Ano v tomto bodě to znamená, že se jednotlivé složky s experimenty mohou nahrávat na testovacím stroji několik hodin, ale jak již bylo řečeno jde o jednorázový úkon.

5.3.6 Tvorba spustitelného JAR souboru aplikace a předání parametru

Jak již bylo zmíněno výše, spustitelný soubor JAR se všemi externími knihovnami vyrobíme pomocí SBT pluginu s názvem Assembly. Ve vývojovém prostředí IntelliJ IDEA je k dispozici interaktivní konzole v záložce s názvem "sbt shell". Tato konzole je po jejím zobrazení automaticky nastavena na aktuální projekt, a jediné co stačí do konzole napsat a klávesou enter spustit je příkaz assembly. Na obrázku níže je zobrazeno zadávání příkazu assembly a textový výpis průběhu jeho zpracování.



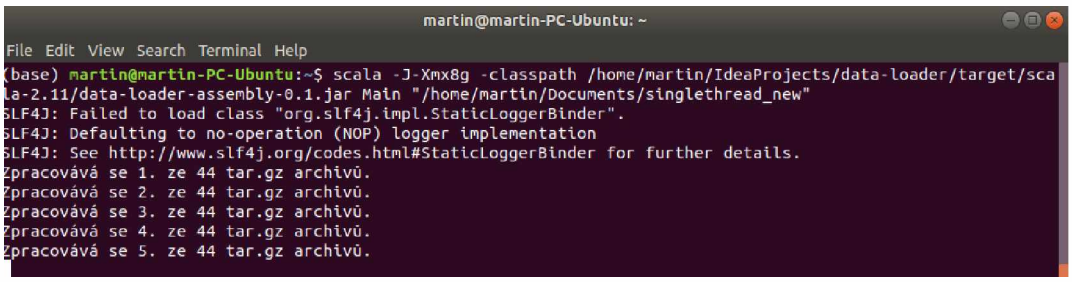
Obrázek 21 – SBT konzole v IDE

Za předpokladu úspěšného vygenerování JAR souboru můžeme přikročit k jeho spuštění. Před spuštěním výsledného konzolového programu je třeba mít na paměti, že příslušné DB tabulky

ve správné databázi musí být už vytvořeny. Dále je třeba při spuštění předat správnou cestu k adresáři s TGZ archivy experimentů z dané skupiny. Argumenty, s kterými je aplikace spuštěna jsou standartně konzumovány metodou main. Přiklad spuštění výsledné aplikace z terminálu je k dispozici níže.

scala -J-Xmx2g -classpath /home/cesta_k_jar_souboru/data-loader-assembly-0.1.jar Main "/home/cesta_ke_slozce_s_archivy_experimentu"

Parametr -J-Xmx2g udává velikost přidělené operační paměti pro JVM. V našem případě je hodnota nastavena na 2 GB, která je plně dostačující pro zpracovávání archivů v operační paměti. Následuje cesta k sestavenému JAR souboru, který chceme spustit, dále je zde jméno souboru se vstupní main metodou a jako poslední je předána cesta k adresáři z kterého chceme nahrát data do DB. Aplikace nás upozorní, pokud jsme parametr se složkou nepředali, popřípadě pokud se v daném adresáři nenachází žádné zpracovatelné TGZ archivy, dále aplikace informuje o tom, který TGZ archiv z kolika se aktuálně zpracovává. Při úspěšném zpracování všech TGZ archivů nás aplikace rovněž informuje a ukončí se. Ukázka běhu programu je na obrázku níže.

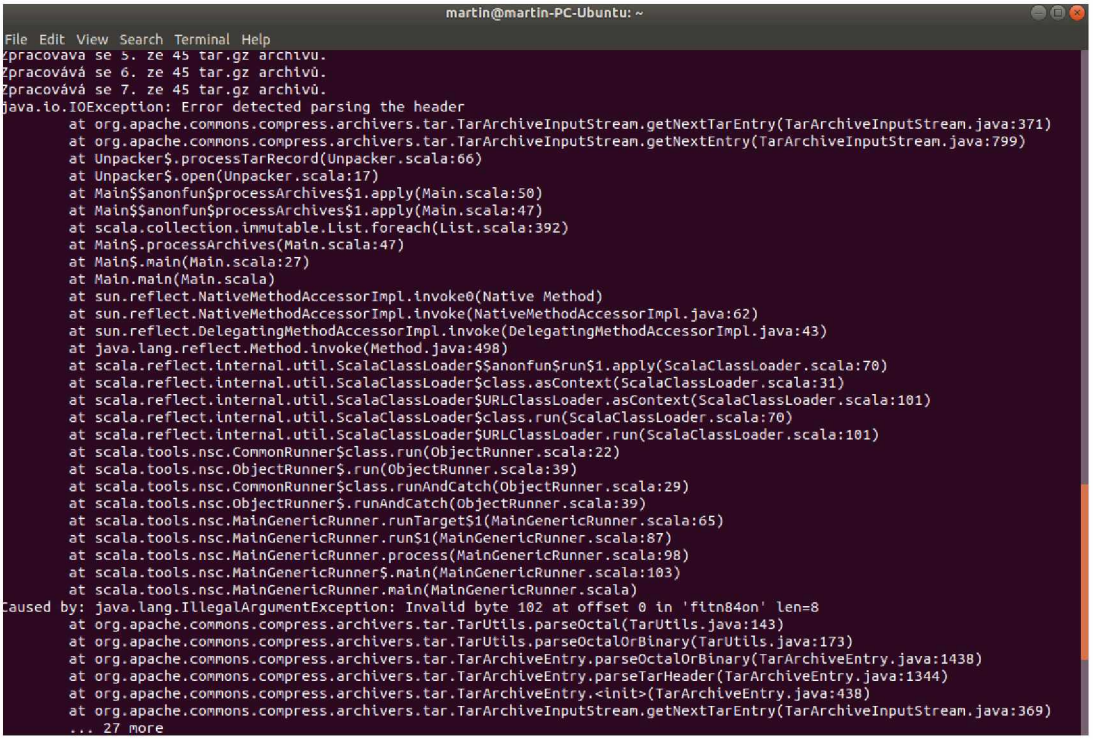


Obrázek 22 – Ukázka z běhu aplikace pro nahrání dat do DB

Při běhu programu je na prvních třech řádcích informační hláška o nepřítomnosti implementace SLF4J. Knihovna ScalikeJDBC umí s výhodou využít implementaci SLF4J pro logování různých nastalých událostí, jako chyb při dotazech nebo vykonané SQL dotazy a podobně. Z důvodu výkonu a také z důvodu, že se nejedná o žádnou aplikaci, která bude někde nasazena a dlouhodobě běžet toto logovaní není třeba a bylo by spíše na obtíž. Případné chyby se vypíší přímo do konzole, z které je aplikace spuštěna, a záznam o proběhlých DB dotazech nepotřebujeme, navíc by se jednalo o obří textový soubor pravděpodobně v řádech GB.

5.3.7 Problémy při nahrávání dat

Při nahrávání dat bylo zjištěno, že některé archivy jsou poškozené a nejdou extrahovat. Tyto archivy s výsledky experimentů byly vyřazeny. Příklad chyby, jež může vzniknout při zpracovávání poškozeného archivu je na obrázku níže.



Obrázek 23 – Chyba při nahrávání dat

Dále se v některých archivech objevují prázdné gpa soubory. Při snaze přečíst první řádek takového souboru samozřejmě aplikace skončila chybou. Jako řešení byla přidána podmínka, zda je první řádek přítomen, pokud není, zpracování takového gpa souboru je přeskočeno.

5.4 Načtení dat v prostředí s pomocí Scala Spark frameworku

V této fázi jsou data z jednotlivých TGZ archivů uložena v DB. Nyní je třeba umět s těmito daty pracovat. Výstupem aplikace, na které vyzkoušena práce s daty z DB, je jakýsi přehled o tom, jaká data máme k dispozici, kolik bylo zpracováno souborů, které hodnoty chybí, popřípadě kolik iterací genetických algoritmů nedoběhlo do konce.

Výstupem této části je Scala Spark aplikace spustitelná přes rozhraní spark-submit.

Celkově data v celém DB schématu zabírají okolo 126 GB. V tabulce níže najdeme přesné počty záznamů v jednotlivých tabulkách.

Tabulka	Počet záznamů	Velikost tabulky [B]
experiment	632	81920
gpa_file	9189304	866123776
gpa_file_row	762744220	119165419520

Tabulka 6 – Velikosti tabulek v DB po nahrání testovaných dat

Z dat vyplývá, že poslední tabulka s názvem gpa_file_row je s bezmála 763 milióny záznamů největší, zabírá okolo 99 procent velikosti celé databáze. Vzhledem k této velikosti je třeba pracovat se záznamy z této tabulky pracovat trochu jinak než se záznamy z tabulek experiment a gpa_file.

5.4.1 Spuštění Spark aplikace pomocí spak-submit

Spuštění Spark aplikace lze provést pomocí spark-submit skriptu který se nachází ve složce bin Spark distribuce. Alternativně je k dispozici spark-shell interaktivní konzole, ale takto vytvořené aplikace slouží spíše pro testování, popřípadě k urychlení vývoje při ověřování konceptu.

Přes spark-submit lze spustit Spark aplikaci v různých módech. Příkaz níže zobrazuje, jak lze spustit aplikaci v lokálním módu, který bude využit při zpracování praktického příkladu.

```
./bin/spark-submit \
  --class cesta.a.nazev.souboru.s.main.metodou.NazevObjektu \
  --master local[*] \
  cesta_k_jar_souboru/sestaveny_jar_soubor_scala_spark_aplikace.jar
```

Povinný je parametr --class, který říká, v jakém souboru se nachází metoda main. Tečky v cestě k souboru představují strukturu jednotlivých balíčků, v kterých může být soubor s main metodou zanořen. Parametrem --master se nastavuje master URL ke clusteru (místo kde Spark aplikace poběží). Pro spuštění na lokálním stroji zde stačí nastavit local, číslo v hranatých závorkách indikuje počet vláken, ve kterých zpracování poběží, pokud je zde místo čísla uvedena hvězdička, využije aplikace všechny logická jádra procesoru, které jsou na daném stroji k dispozici. Pokud bychom chtěli zpracovávat data na existujícím Spark clusteru obsahuje tato hodnota URL adresu tohoto clusteru. K dispozici jsou prefixy spark://, mesos:// a k8s:// pro Spark samostatný cluster, Mesos cluster, respektive Kubernetes cluster. V neposlední řade zde může být hodnota yarn pro připojení k YARN clusteru.

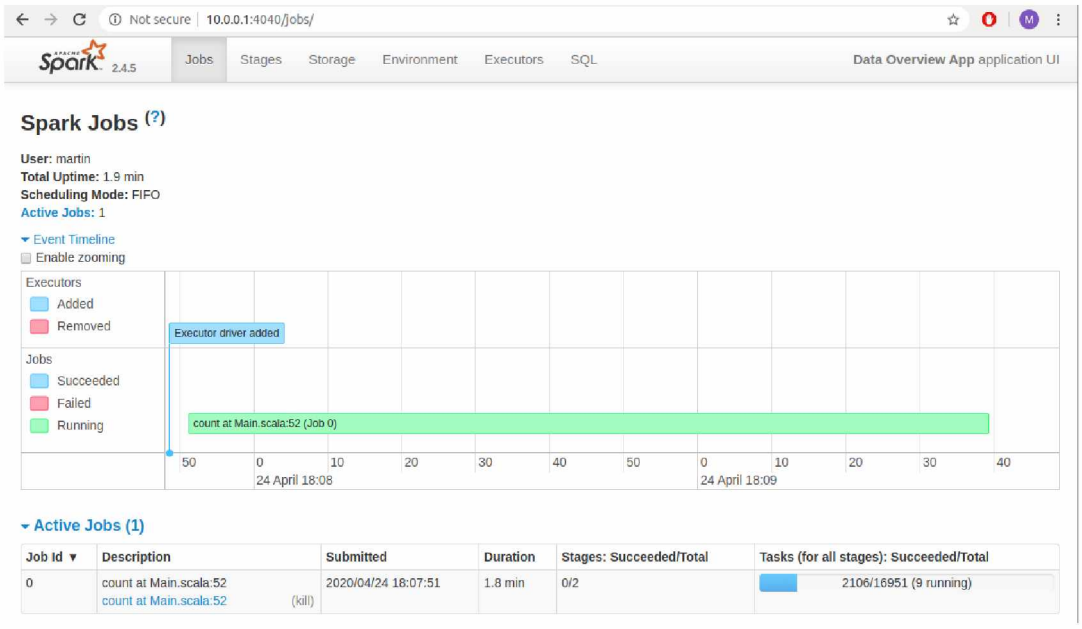
V příkladu byly uvedeny jen parametry master a class, jelikož tyto parametry stačí k lokálnímu spuštění, nicméně v praxi se setkáme s dalšími důležitými parametry jako třeba debloy-mode. Tento parametr může nabývat hodnot client nebo cluster, defaultně je to hodnota client. V módu client běží Spark ovladač lokálně jako externí klient. Toto se hodí zejména pro debuggování, kdy chceme okamžitě vidět výsledky. Alternativně v módu cluster běží Spark ovladač na jednom z uzlů clusteru, to se hodí především v produkčním prostředí. Další parametry slouží k nastavení operační paměti, kterou bude Spark využívat. Parametrem driver-memory lze nastavit velikost paměti pro Spark ovladač, executor-memory zase nastaví velikost jednotlivým

vykonavačům. V lokálním módu máme jen ovladač, není tedy třeba nastavovat vlastnost executor-memory. Na druhou stranu driver-memory je dobré nastavit, doporučeno je okolo 75% celkové paměti která je k dispozici. Zbytek paměti je dobré nechat pro operační systém a vyrovnávací paměť. Pro zpracování praktické úlohy byl využit počítač s 32 GB operační paměti. Pro potřeby Spark ovladače bylo tedy přiřazeno 24 GB. Reálné spuštění skriptu v lokální módu tedy může vypadat jako na ukázce níže.

```
./spark-submit \
  --driver-memory 24G \
  --class Main --master local[*] \
  /home/martin/IdeaProjects/spark-data-overview/target/scala-2.11/spark-data-overview-assembly-0.1.jar
```

5.4.2 Monitorování běhu Spark Aplikace

V lokálním módu lze při defaultním nastavení logování sledovat průběh prováděných operací pomocí výpisu přímo v konzoli, ze které byla Spark aplikace spuštěna. Toto ovšem nemusí být úplně přehledné a uživatelsky přívětivé. Z tohoto důvodu nabízí Spark webové rozhraní, ve kterém lze sledovat průběh právě prováděné úlohy. K dispozici je zde mimo jiné seznam dostupných vykonavačů a jejich vlastnosti. K právě prováděné úloze si je možné zobrazit DAG plán operaci nad jednotlivými RDD. K dispozici je i záložka SQL, která nabízí detailnější pohled nad dotazy, které probíhají v rámci operací zpracování pomocí Spark SQL modulu. Ukázka právě zpracovávané úlohy je k dispozici na obrázku níže.



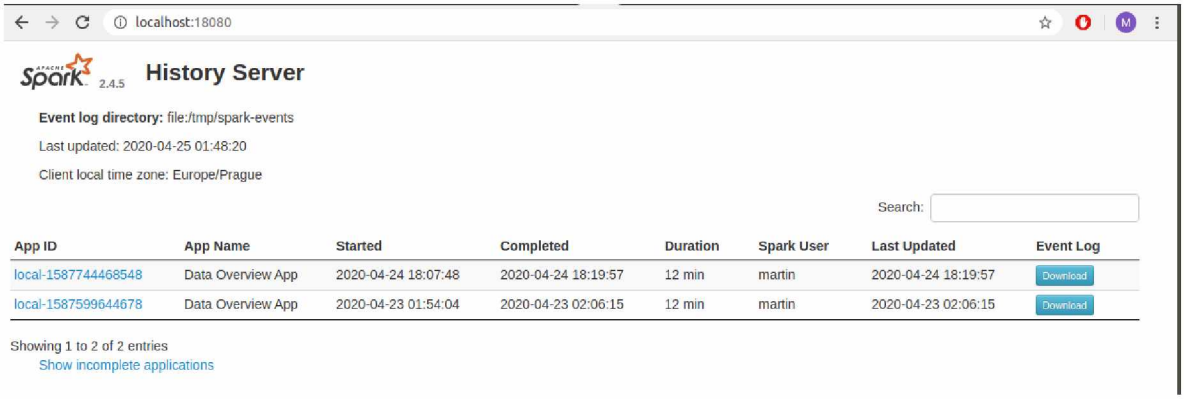
Obrázek 24 –Monitorování běhu Spark aplikace

Z ukázky je patrné, že Spark webové rozhraní se nachází na adrese 10.0.0.1, což je lokální adresa, na které běží Spark ovladač. Webové rozhraní spouští automaticky objekt SparkContext a výchozí port má číslo 4040. Pokud by tento port byl obsazený, pokusil by se Spark využít port 4041, 4042 atd. Při spuštění Spark úlohy je adresa, na které toto rozhraní vypsána.

Pokud je třeba analyzovat již zpracovanou úlohu, i zde existuje řešení. Spark si může ukládat záznamy o vykonaných úlohách, pro povolení ukládání záznamů o zpracovaných úlohách je třeba nastavit parametr spark.eventLog.enabled na hodnotu true. S tímto ukládáním souvisí ještě další konfigurační parametry. Nejdůležitější z nich je spark.eventLog.dir který říká, na jaké místo se jednotlivé logy ukládají. Pro prohlížení těchto historických záznamů je třeba spustit server který to umožní. V složce sbin se ve Spark distribuci nachází pro tento účel skript start-history-server.sh. Po spuštění tohoto skriptu se server nastartuje. Server používá konfigurační parametr spark.history.fs.logDirectory, který musí být nastaven na místo kde se nachází historické záznamy s běhy Spark aplikací. Zjednodušeně by měl většinou odpovídat parametru spark.eventLog.dir. Aby se server spustil úspěšně je třeba aby nastavená lokace skutečně existovala, tzn. v lokálním prostředí vytvořit danou složku předem. Defaultní hodnota pro parametr spark.history.fs.logDirectory je file:/tmp/spark-events, po vytvoření této složky nám nic nebrání v tom spouštět Spark aplikace s konfiguračním spark.eventLog.enabled nastaveným na true, defaultní umístění ukládání záznamů je shodné s lokací kam kouká defaultně server pro zobrazení historických záznamů. Spuštění aplikace se zapnutým logováním je na ukázce níže.

```
./spark-submit
  --conf spark.eventLog.enabled=true
  --class Main --master local[*]
  /home/martin/IdeaProjects/spark-data-overview/target/scala-2.11/spark-data-overview-assembly-0.1.jar
```

Po dokončení zapracování dat je záznam z běhu aplikace k dispozici pro analýzu ve webovém rozhraní jako je to na obrázku níže.



Obrázek 25 – Monitorování již doběhnutých Spark aplikací

Webové rozhraní je prakticky totožné jako v případě kdy úloha stále běží, s tím rozdílem, že na začátku je nutno vybrat již zpracovanou úlohu, jelikož jich může být ve složce více. Ke skriptu start-history-server.sh je k dispozici protějšek stop-history-server.sh, který server s historickými záznamy vypne.

5.4.3 Tvorba Scala Spark Aplikace

Podobně jako při ukládání dat je následující projekt SBT aplikací sestavenou Assembly pluginem. Nemusíme se tedy při spuštění Spark aplikace zabývat předáváním externích knihoven, ať už předaním samotných souborů nebo předáním Maven uložiště s popisem odkud co stáhnout. Struktura projektu standartní a podobná předchozí aplikaci. V ukázce níže je obsah souboru build.sbt.

```
name := "spark-data-overview"
version:= "0.1"
scalaVersion:= "2.11.12"

libraryDependencies += "org.apache.spark" %% "spark-sql" % "2.4.5" % "provided"
libraryDependencies += "org.mariadb.jdbc" % "mariadb-java-client" % "2.5.2"
libraryDependencies += "org.scalikejdbc" %% "scalikejdbc" % "3.4.0"
libraryDependencies += "org.scalikejdbc" %% "scalikejdbc-config" % "3.4.0"
```

U knihovny spark-sql zde přibyl definovaný rozsah. Rozsah, který je u spark-sql knihovny nastaven jako “provided”, neboli poskytnutý indikuje, že prostředí, ve kterém budeme tuto aplikaci spouštět tuto knihovnu obsahuje, knihovna se tedy nemusí balit společně s ostatními externími knihovnami. Toto řešení je výhodné z hlediska velikosti výsledného JAR souboru. Bez přibalené spark-sql knihovny a veškerých knihoven, na kterých tato knihovna závisí, je výsledná velikost souboru nižší o více než 107 MB.

Po založení projektu je třeba inicializovat objekt SparkSession. SparkSession objekt poskytuje veškeré dostupné Spark API. Tento centralizovaný vstupní bod byl do prostředí Spark

frameworku představen s jeho verzí 2.0. Před verzí 2.0 se jako vstupní body k Spark API používalo více kontextů, hlavní byl vždy SparkContext, díky kterému se daly vytvořit následně další kontexty jako SQLContext nebo například HiveContext. Tento postup byl kvůli zpřehlednění a zjednodušení práce se Sparkem nahrazen. Na ukázce níže je vytvoření objektu SparkSession a také následné ukončení Spark aplikace.

```
val spark = SparkSession.builder.appName("Data Overview App").getOrCreate()
…
spark.stop()
```

Místo tří teček si lze dosadit jakoukoli práci s daty za pomocí Spark API. Po zavolání metody stop na konci aplikace dojde k úklidu po dané aplikaci. Ukončí se veškerá její činnost a smažou se dočasně vytvořené soubory, například cache soubory často využívaných dat a podobně.

V tomto okamžiku tedy máme připravenou strukturu projektu a chceme načíst data z databáze. V ukázce níže je způsob, jakým lze ve Sparku načíst data z DB tabulky.

```
val experimentDF = spark.read
  .format("jdbc")
  .option("url", "jdbc:mysql://localhost:3306/thesis")
  .option("dbtable", "experiment")
  .option("driver", "org.mariadb.jdbc.Driver")
  .option("user", "root")
  .option("password", "password").load.persist();
```

Obdobně jako v předchozí aplikaci je zde opět využit ovladač pro MariaDB. Kvůli problémům vzniklým při mapování výsledků je lepší ve vlastnosti url použít v cestě mysql místo mariadb. Takto načítání záznamů z DB funguje vždy korektně a bez chyby. Samotný kód výše nicméně k samotnému načtení dat z DB nevede. Aby se data z DB skutečně načetla, je nutné DataFrame experimentDF použít. Toho lze například jako v případě níže docílit provoláním metody count která vrátí celkový počet záznamů v tabulce experiment.

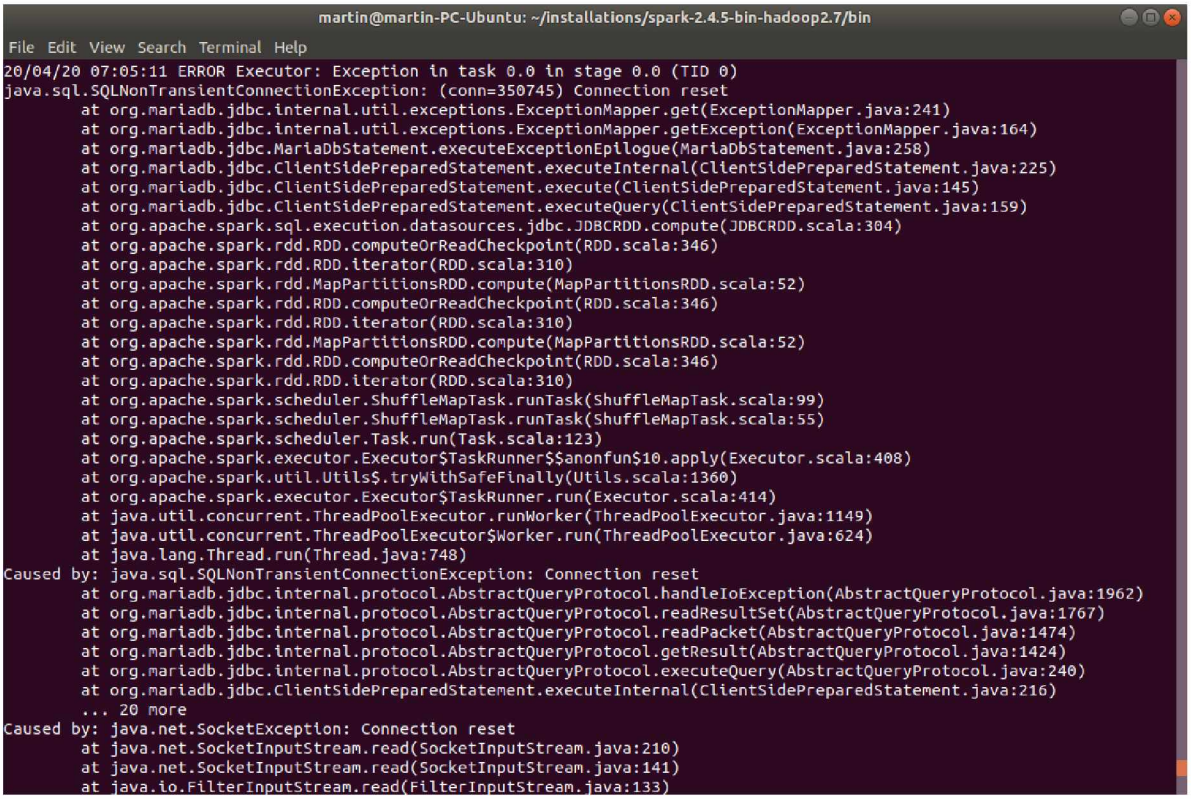
```
println(s"Počet záznamů v tabulce experiment je: ${experimentDF.count()}")
```

Pro zjednodušení práce autor při tvorbě všech Spark aplikací napsal přihlašovací údaje a jméno databáze přímo do zdrojového kódu, pokud tedy přesně nesedí tyto údaje s databází, z které chceme data experimentů načítat, je třeba tyto hodnoty přepsat a aplikaci překompilovat. Nicméně jelikož jsou výsledné parquet soubory distribuovány s diplomovou prací není třeba tyto aplikace spouštět.

Pokud bychom chtěli načíst data pomocí SQL dotazu, lze použít místo vlastnosti dbtable vlastnost query. Příklad s vlasností query je k nahlédnutí níže.

```
val experimentDF = spark.read
  .format("jdbc")
  .option("url", "jdbc:mysql://localhost:3306/thesis")
  .option("query", "select id, rand from experiment")
  .option("driver", "org.mariadb.jdbc.Driver")
  .option("user", "root")
  .option("password", "password").load
```

Výhoda takového postupu je možnost už zde rovnou provést selekci sloupců s kterými chceme pracovat a které nás zajímají. V příkladu byly vybrány sloupce id a rand. Obdobně lze načíst tabulku gpa_file. V případě tabulky gpa_file_row nicméně dostaneme pouze chybu jako na obrázku níže.



Obrázek 26 – Exspirované připojení do DB

Tato chyba říká, že během zpracování SQL dotazu došlo k restartovaní spojení. Důvodem tohoto chování je příliš velká tabulka. Databáze se dlouho Sparku nehlásí a ten spojení přeruší. Určitým konfiguračním parametrem lze tuto dobu čekání na odpověď prodloužit, ale ani to není správná cesta v případě tabulky která obsahuje přes 100 GB dat. Správná cesta je načítat data po částech. Tabulku je nutné rozdělit na části, které lze samostatně načíst podstatně rychleji než celou tabulku. Spark samozřejmě s takto velkými tabulkami počítá a umožnuje jednoduše načítat data po částech. Příklad takového načítání je k dispozici v ukázce níže.

```
val gpaFileRowDF = spark.read
  .format("jdbc")
  .option("url", "jdbc:mysql://localhost:3306/thesis")
  .option("dbtable", "thesis.gpa_file_row")
  .option("driver", "org.mariadb.jdbc.Driver")
  .option("user", "root")
  .option("lowerBound", min)
  .option("upperBound", max)
  .option("numPartitions", numPartitions)
  .option("partitionColumn", "gpa_file_row_id")
  .option("password", "password").load;
```

Tabulku rozdělíme na části podle primárního klíče, který má celočíselný datový typ, což je přesně to, co je zde potřeba. V načítaní přibyly čtyři nové vlastnosti, první z nich je partitionColumn, udávající sloupeček, podle kterého se tabulka bude dělit. Dále přibyly vlastnosti loverBound a upperBound, což je dolní, respektive horní hranice primárního klíče, podle kterého probíhá dělení. Poslední vlastností je numPartitions, jež udává počet bloků, které se budou zvlášť načítat. V následné ukázce je způsob, jakým lze získat hodnoty lowerBound, upperBound a numPartitions v závislosti na velikosti bloku, který je zde nastaven na 50000 záznamů.

```
case class Table(min: String, max: String)
val table: Option[Table] = sql"select min(gpa_file_row_id) as min,
max(gpa_file_row_id) as max from gpa_file_row"
  .map(rs => Table(rs.string("min"), rs.string("max"))).single.apply()

val min = table.get.min.toInt
val max = table.get.max.toInt
val numPartitions = (max - min) / 50000 + 1
```

Pro optimální rychlost zpracování dotazu je třeba najít optimální velikost načítaného bloku. Z tohoto důvodu bylo načítání výsledků z tabulky gpa_file_row provedeno několikrát. Optimální dobu zpracování pro danou velikost bloku lze vyčíst z tabulky níže.

Velikost bloku	10000	45000	50000	55000	100000
Doba zpracování [s]	1748.28	871.43	841.84	1040.79	1250.89

Tabulka 7 – Doba zpracování v závislosti na velikosti bloku

Z tabulky vyplývá, že optimální velikost bloku pro tabulku gpa_file_row v testovacím prostředí je okolo 50000. Tato hodnota se může lišit v závislosti na složitosti jednoho záznamu, kvality připojení Sparku do DB, a hlavně pak na rychlosti samotné DB. V případě, kdy databáze byla nainstalována na rychlejším SSD disku v prostředí Windows, byla optimální velikost bloku okolo 2300000, a to i přestože DB tabulka byla identická a Spark rovněž běžel lokálně na stejném stroji jako nainstalovaná databáze. Samotný disk v prostředí Windows má rychlost čtení

trojnásobně rychlejší než SSD disk z prostředí Linux, rozdíl v ideální velikosti bloku byl skutečně markantní, daleko vetší než trojnásobný.

5.5 Spark aplikace pro analýzu dat

Jak již bylo řečeno pro analýzu dat vznikly celkem tři Scala Spark aplikace, při kterých byly použity poznatky a metody z předchozích kapitol. Ke každé Spark aplikaci byl posléze vytvořen sešit v sešitovém rozhraní Zeppelin. Výčet aplikací se základním popisem lze nalézt v tabulce níže.

Aplikace	Funkce
spark-data-overview	Poskytnout základní přehled o datech která jsou k dispozici. Například počet dat pro jednotlivé skupiny testů a tak podobně.
spark-results-aggregation	Nabízí ucelenější pohled na výsledky a ukazuje práci s některými statistickými nástroji které jsou k dispozici ve frameworku Spark.
spark-histograms	Hlavní úkolem je vygenerovat data ze kterých se pomocí knihovny Bokeh vykreslí histogramy. Dále obsahuje některé další statistické nástroje a v neposlední řadě také funkci jež ohodnocuje výsledné řešení dle složitosti výstupních rovnic.

Tabulka 8 – Spark aplikace které vznikly při analýze vzorové sady dat

Všechny tři aplikace generují parquet soubory, které budou později načteny a graficky zobrazeny v již zmíněném Zeppelin sešitu. Jednotlivé parquet soubory s popisem toho, co jejich data obsahují, generované z aplikace spark-data-overview jsou k dispozici v tabulce níže.

generalOverview1	Počet gpa souborů a jejich členů populace v nich obsažených dělených podle skupin a kombinace generátorů.
generalOverview2	Počet experimentů k dispozici pro skupinu a kombinaci generátorů.
generalOverview3	V návaznosti na předchozí parquet přidává dodatečnou informaci o speciálních kombinacích generátorů.
generalOverview4	Pohled na očekávaný počet gpa souborů a jejich skutečný počet.

generalOverview5	Počet adresářů s konkrétním počtem gpa souborů v nich obsažených.

Tabulka 9 – Přehled parquet souboru z aplikace spark-data-overview

Další tabulka níže zobrazuje parquet soubory z aplikace spark-results-aggregation.

resultsAggregation1	Experimenty s žádným gpa souborem, kde zpracování úplně selhalo nebo nebylo provedeno.
resultsAggregation2	V návaznosti na předchozí parquet soubor poskytuje detailnější informace o těchto experimentech.
resultsAggregation3	Nalézá poměr gpa souborů k počtu gpa souborů s nalezeným řešením pro jednotlivé experimenty.
resultsAggregation4	Stejná data jako u předchozího parquet souboru, ale tentokrát pro celé skupiny experimentů.
resultsAggregation5	Pro jednotlivé experimenty poskytuje statistické ukazatele v podobě průměru, směrodatné odchylky, minimální a maximální hodnoty a mediánu pro počet cyklů a délku výpočtu.
resultsAggregation6	Pro jednotlivé experimenty tento parquet soubor nabízí percentily různých úrovní pro počet cyklů.
resultsAggregation7	Stejný jako předchozí parquet soubor, nicméně s percentily pro délku hledání řešení.

Tabulka 10 – Přehled parquet souboru z aplikace spark-results-aggregation

Poslední tabulka parquet souboru pak zobrazuje níže soubory z aplikace spark-histograms.

covariance_correlation	Kovariance a korelace mezi číslem iterace sloužícím jako semínko generátoru náhodných čísel a počtem cyklů.
cycles_histogram	Hodnoty potřebné pro konstrukci histogramu, v tomto případě pro počet cyklů.
cycles_histogram_filtered	Stejný jako předchozí parquet soubor s tím rozdílem, že neobsahuje nedokončené iterace a jejich data.

duration_histogram	Hodnoty potřebné pro konstrukci histogramu, v tomto případě pro délku výpočtu.
duration_histogram_filtered	Stejný jako předchozí parquet soubor s tím rozdílem, že neobsahuje nedokončené iterace a jejich data.
formula_complexity	Parquet soubor s ohodnoceným řešením podle počtu závorek ve výsledné rovnici.

Tabulka 11 – Přehled parquet souboru z aplikace spark-histograms

V následující sekci je zobrazeno, jakým způsobem lze v aplikaci Spark pracovat s daty pomocí Spark SQL API a jak lze výsledné objekty typu DataFrame uložit na disk. V následující ukázce níže je pokryt jen jeden z mnoha parquet souborů, které při práci na diplomové práci vznikly.

```
experimentJoinedWithGpaFile
  // vyfiltrovat nedopocitane gpa soubory
  .filter(col("cycle").lt(col("no_populations")))
  // vyfiltrovat experimenty s malym poctem iteraci
  .filter("experiment_id != 589 AND experiment_id != 568")
  .groupBy("folder", "rand", "experiment_id")
  .agg(
    stddev("cycle").as("stddev_cycle"),
    stddev("duration").as("stddev_duration"),
    mean("cycle").as("mean_cycle"),
    mean("duration").as("mean_duration"),
    percentile_approx(col("cycle"), lit(0.25)).as("cycle_percentile_0_25"),
    percentile_approx(col("duration"), lit(0.25)).as("duration_percentile_0_25"),
    percentile_approx(col("cycle"), lit(0.5)).as("cycle_percentile_0_5"),
    percentile_approx(col("duration"), lit(0.5)).as("duration_percentile_0_5"),
    percentile_approx(col("cycle"), lit(0.75)).as("cycle_percentile_0_75"),
    percentile_approx(col("duration"), lit(0.75)).as("duration_percentile_0_75"),
    percentile_approx(col("cycle"), lit(0.9)).as("cycle_percentile_0_9"),
    percentile_approx(col("duration"), lit(0.9)).as("duration_percentile_0_9"),
    percentile_approx(col("cycle"), lit(0.99)).as("cycle_percentile_0_99"),
    percentile_approx(col("duration"), lit(0.99)).as("duration_percentile_0_99")
  ).write.mode(SaveMode.Overwrite).parquet("/tmp/output/resultsAggrega-
tion6.parquet")
```

Ukázka pochází ze Spark aplikace spark-results-aggregation. Pro analýzu zde posloužil DataFrame objekt experimentJoinedWithGpaFile jež jak název napovídá obsahuje spojené záznamy z tabulek experiment a gpa_file. Povšimnout si lze přehledného API objektu DataFrame. V podstatě každý, kdo se dříve setkal s jazykem SQL, okamžitě pochopí, co se děje. Nejdříve je provedeno vyfiltrování dat, která jsou pro analýzu nevhodná, tzn. gpa souborů u kterých zpracování nedoběhlo do konce. Následně je třeba ještě odstranit data z experimentů s identifikátory 589 a 568, a to z důvodu nedostatku iterací v rámci experimentu. Takováto data by jednoduše nebyla věrohodná. Následuje rozdělení dat na skupiny podle sloupců folder, rand a experiment_id. S tímto rozdělením můžeme použít agregační funkce, které pro tyto vytvořené

skupiny mohou spočítat průměr, směrodatnou odchylku nebo vybrané percentily. Na závěr už je třeba jen uložit výsledná data z analýzy do parquet souboru na vybranou lokaci na disku.

Obdobně jako byl vytvořen předchozí parquet byly zkonstruovány všechny výstupní parquet soubory všech tří Spark aplikací. Spark samozřejmě není limitován jen existujícími agregačními funkcemi, exituje možnost si napsat vlastní funkce, popřípadě pracovat s daty mimo standartní API, pořád jsou zde nekonečné možnosti jazyka Scala, a tak jediným limitem je badatelova představivost. Agregační funkce percentile_approx je ve skutečnosti taky uživatelsky napsaná funkce, neboť se ve veřejném API nenachází, její implementace je v ukázce níže.

```
object PercentileApprox {
  def percentile_approx(col: Column, percentage: Column, accuracy: Column): Column
= {
    val expr = new ApproximatePercentile(
      col.expr,  percentage.expr, accuracy.expr
    ).toAggregateExpression
    new Column(expr)
  }
  def percentile_approx(col: Column, percentage: Column): Column = percentile_ap-
prox(
    col, percentage, lit(ApproximatePercentile.DEFAULT_PERCENTILE_ACCURACY)
  )
}
```

Obdobně jako lze definovat agregační funkci lze také definovat UDF, neboli uživatelky definovanou funkci. Příklad funkce, která ze sloupce folder podle názvu vrací počet předpokládaných unikátních gpa složek je k dispozici níže.

```
def getExpectedGpaFolders = udf {
  (folder: String) => {
    if (folder == "singlethread_old") 2000 else 10000
  }
}
```

Pro další příklady práce s DataFrame API je možné nahlédnout do zdrojového kódu aplikací, který je distribuovaný společně s diplomovou prací, popřípadě lze například nahlédnout do Spark dokumentace.

5.6 Práce s Apache Zeppelin

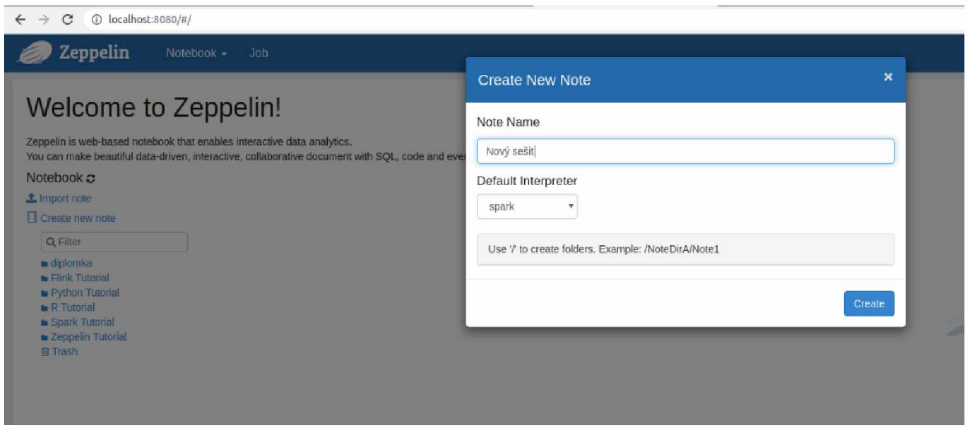
Po zpracování dat pomocí frameworku Apache Spark máme k dispozici výsledné parquet soubory. Pro další práci je třeba zvolit vhodný nástroj s pomocí kterého lze výsledná data graficky zobrazit a tím snadněji vyhodnotit. Tímto nástrojem se ukázal být Apache Zeppelin. V několika následujících odstavcích bude předvedeno, jak lze s tímto sešitovým rozhraním pracovat.

5.6.1 Práce s interprety

Interpret v prostředí Apache Zeppelin slouží k ovládání dané technologie. Pod tím si lze představit třeba interpret pracující s Apache Spark nebo například takřka jakoukoliv databází. Pro Spark je k dispozici interpretů více. Existuje interpret pro práci se Scala Sparkem, stejně tak existuje interpret pro práci s Python Sparkem a ještě další, z nichž jeden bude použit v ukázce práce s Bokeh knihovnou.

Celkem v průběhu práce vznikly tři Zeppelin sešity. Každý jeden sešit reprezentuje dříve zmíněné Spark aplikace a zobrazuje data pouze z korespondujících vygenerovaných parquet souborů.

V ukázce je způsob, jakým lze vytvořit nový sešit v uživatelském prostředí Zeppelin.

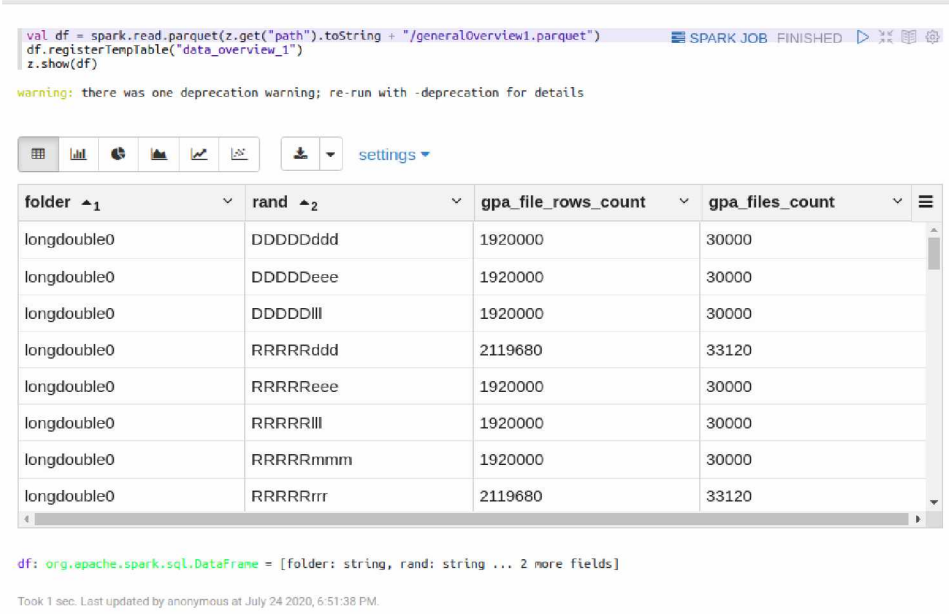


Obrázek 27 – Vytváření nového Zeppelin sešitu

Pro nově vytvořený sešit s názvem Nový sešit byl zvolen defaultní interpret spark. To znamená, že při vytvoření nového odstavce v tomto sešitu není třeba specifikovat, že uživatel chce pracovat s technologii Spark. Zeppelin umožňuje pracovat s více technologiemi v rámci jednoho sešitu. Toto lze specifikovat definováním jiného interpretu na začátku odstavce ve formátu %interpret.

5.6.2 Načtení parquet souborů v Zeppelin sešitu

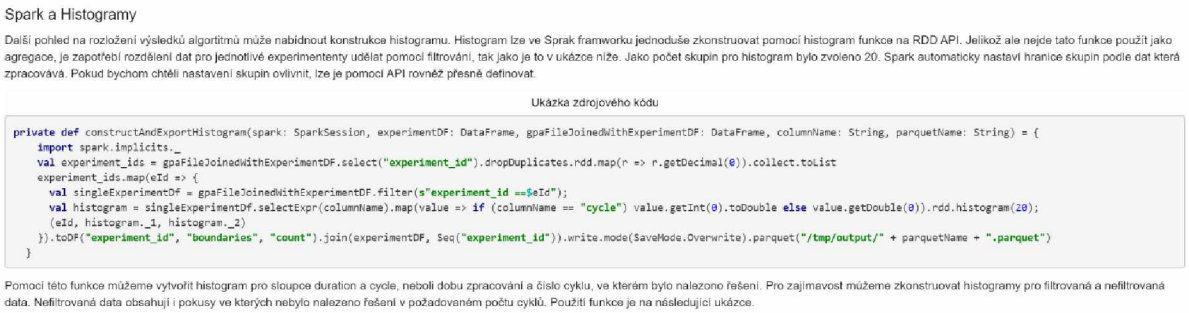
V následující sekci je ukázáno, jakým způsobem lze načíst data z parquet souborů za pomocí Spark interpretu. Postup je velice jednoduchý, a z načteného objektu DataFrame lze bez jakéhokoliv dalšího programování zobrazit graf či tabulku pomocí automaticky vytvořeného objektu z a jeho metody show, jako je to zobrazeno v ukázce níže.



Obrázek 28 – Integrované vizualizační nástroje dostupné v Zeppelinu

Z obrázku je patrné, že samotné načtené parquet souboru je záležitost jednoho řádku kódu, poté už si lze z nabídky vybrat vhodný typ grafu či tabulkovou reprezentaci dat.

Zeppelin také umožňuje přidat do výstupu ovládací prvky ať, už různá vyhledávací políčka či jiné nabídky. Toho lze využít zejména pro prezentaci dat, popřípadě vytvořit uživatelské rozhraní pro někoho bez znalosti programování a tak podobně. Zeppelin sešit má taky místo editačního módu k dispozici mód pro reporting, v takovém případě veškerý kód, kterým jsou definovány jednotlivé odstavce zmizí a je k dispozici nerušený pohled na tabulky a grafy, popřípadě text s bližším vysvětlením. Pro jakékoliv bližší vysvětlení lze použít jazyk HTML. Za pomocí HTML byly v diplomové práci vytvořeny bloky se zdrojovým kódem zpracovávaných parquet souborů, ukázku takového bloku lze vidět na obrázku níže.



Obrázek 29 – Ukázka zdrojového kódu v Zeppelin sešitu

V další obrázku níže je ukázka dříve zmíněných interaktivní prvků Zeppelinu. Konkrétně je zde zobrazena tabulka s nabídkou podle čeho se má tato tabulka řadit. Toto řešení bylo implementováno kvůli chybám v Zeppelin řazení, které ne vždy fungovalo korektně, s implementovaným řešením se data třídí přímo pomocí Sparku.

```
import org.apache.spark.sql.{DataFrame}

val select = z.select("Seřadit dle",Seq(("1", "no_populations vzestupně"),
                                        ("2","no_populations sestupně"),
                                        ("3","finished_in_time_count vzestupně"),
                                        ("4","finished_in_time_count sestupně"),
                                        ("5","gpa_files_count vzestupně"),
                                        ("6","gpa_files_count sestupně"),
                                        ("7","succesfully_finished_ratio vzestupně"),
                                        ("8","succesfully_finished_ratio sestupně")
                                       ), "7")


val df = spark.read.parquet(z.get("path").toString + "/resultsAggregation3.parquet")
df.registerTempTable("results_aggregation_3")
z.show(sortDataFrame(df))


def sortDataFrame =
    (df: DataFrame) => {
      select.toString match {
        case "1" => df.orderBy(asc("no_populations"))
        case "2" => df.orderBy(desc("no_populations"))
        case "3" => df.orderBy(asc("finished_in_time_count"))
        case "4" => df.orderBy(desc("finished_in_time_count"))
        case "5" => df.orderBy(asc("gpa_files_count"))
        case "6" => df.orderBy(desc("gpa_files_count"))
        case "7" => df.orderBy(asc("succesfully_finished_ratio"))
        case "8" => df.orderBy(desc("succesfully_finished_ratio"))
      }
    }
```

Seřadit dle

succesfully_finished_ratio vzestupně

```
warning: there was one deprecation warning; re-run with -deprecation for details
```

settings

folder	rand	experiment_id	no_pop
longdouble0	DDDDDeee	20	10000

Obrázek 30 – Interaktivní uživatelské prvky v Zeppelinu

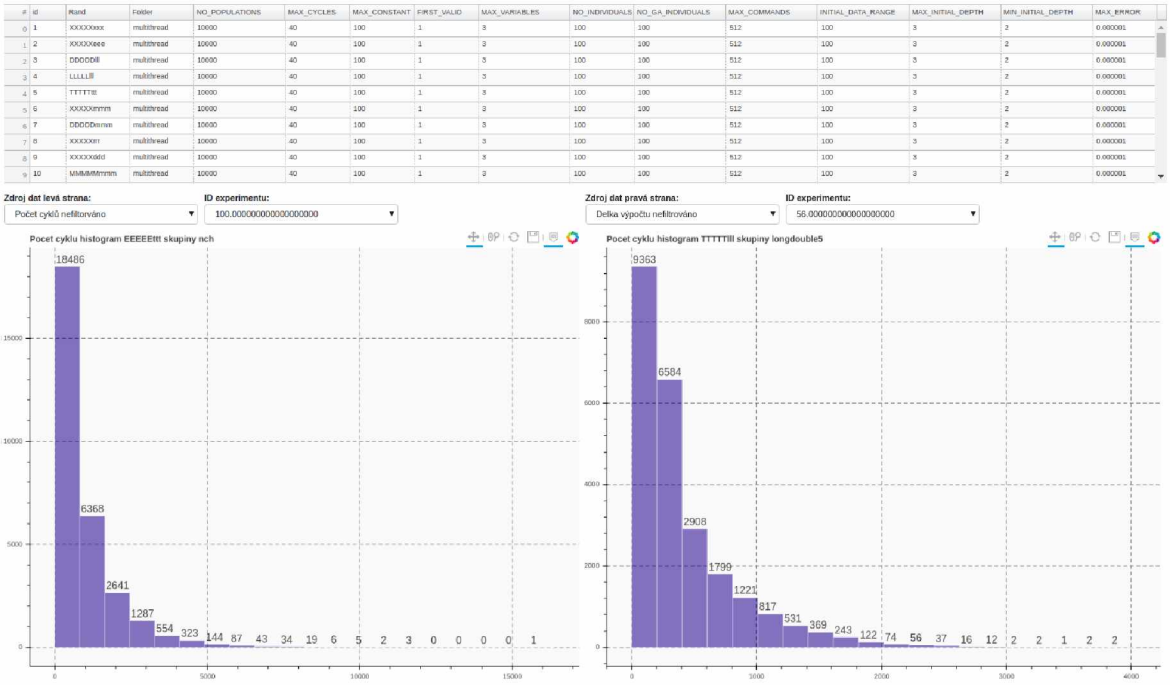
5.7 Tvorba grafů s knihovnou Bokeh v Zeppelin sešitu

Bohužel na některé případy Zeppelin integrované grafy a tabulky svojí omezenou funkcionalitou nestačí. Pro tyto případy byla zvolena knihovna Bokeh. S nástrojem Bkzep lze výstupy vizualizací vykreslovat přímo do Zeppelin sešitu. Bokeh dokonce umožňuje využít vlastního serveru a reagovat tím na interakce uživatele pomocí jazyka Python. Další řešení by bylo zapojení Javascriptu, v takovém případě by nebylo ale možné třeba poslat data zpět do Sparku a podobně. Nalezené řešení pro start serveru s obsluhovačem změn, který se stará interakci uživatele pomocí interaktivních ovládacích prvků je v ukázce níže.

```
%spark.ipyspark

import yaml

from bokeh.plotting import figure
from bokeh.io import show, output_notebook
from bokeh.models import ColumnDataSource, PreText, Select, LabelSet
from bokeh.layouts import column, row

from bokeh.models.widgets import DataTable, TableColumn

from bokeh.themes import Theme
import pandas as pd
import bkzep

output_notebook(notebook_type='zeppelin')


def modify_document(document):
    # handler s obsahem co se ma zobrazit, timto zpusobem lze s daty pracovat v realnem case v pythonu bez nutnosti primo pouzivat javascript

from bokeh.application.handlers import FunctionHandler
from bokeh.application import Application

app = Application(FunctionHandler(modify_document))

show(app, notebook_url="localhost:8080")
```

Obrázek 31 – Kostra pro interaktivní rozhraní prostředí

Bokeh knihovna byla využita v praktické části jen k zobrazení histogramů. Celé vytvořené uživatelské prostředí je zobrazeno na obrázku níže. V případě zájmu je zdrojový kód k této části obsahem Zeppelin sešitu s názvem bokeh_a_histogramy.



Obrázek 32 – Prostředí s histogramy vytvořenými knihovnou Bokeh

5.8 Výstup praktické části

Absolutním výstupem praktické části jsou Zeppelin sešity s interaktivními grafy a tabulkami, které by mohli být zajímavé z hlediska na první pohled skrytých informací. Aby uživatel nemusel sám spouštět samotnou analýzu dat ve Sparku, a tím vlastně i nahrávat zdrojová data do DB jsou výsledná data distribuována ve formě parquet souborů. Nejnutnější minimum pro prozkoumání výsledků je tedy mít nainstalovaný Zeppelin a s ním distribuovaný Spark. V Zeppelinu se potom dají jednotlivé sešity, jež jsou distribuovány s touto prací importovat. Pokut by

uživatel nechtěl instalovat ani Zeppelin jsou k dispozici náhledy ve formě html souborů, nicméně tyto náhledy nejsou interaktivní ale jen statické, nelze například posouvat data v tabulkách či měnit data grafů. Každý výsledný Zeppelin sešit obsahuje odstavec, ve kterém je definována cesta k adresáři s výslednými parquet soubory. Tuto cestu si uživatel může přenastavit podle toho, kde se soubory nachází na jeho počítači, defaultní hodnota je /tmp/output. Alternativně by šla práce distribuovat i na službách, jež nabízejí hosting pro Zeppelin sešity, v takovém případě by nebylo potřeba na klientském počítači instalovat dodatečný software.

Ve výsledných Zeppelin sešitech bylo například zjištěno následující:

- K dispozici je celkem devět skupin experimentů s různým počtem kombinací náhodných generátorů. Nejvíce kombinací náhodných generátorů, konkrétně 252 je k dispozici pro skupinu označenou jako singlethread_old, nejméně pak pro skupinu označenou jako longdouble10 a to 15.
- Co se týče kombinací generátorů náhodných čísel existuje jen jedna kombinace, pro kterou jsou k dispozici výsledky testů v rámci všech skupin experimentů a tou je kombinace generátorů s názvem RRRRRrrr. Naopak existuje velké množství kombinací které jsou pouze pro skupinu s názvem singlethread_old.
- Nejvíce řádků v gpa souborech neboli záznamů o členech populace, je opět k dispozici pro skupinu s názvem singlethread_old, konkrétně 227 milionů. Nejméně pak pro skupinu longdouble0 a to 31 milionů. Překvapivě to není skupina longdouble10, která má nejméně kombinací náhodných generátorů čísel.
- Pro některé kombinace náhodných generátorů je v rámci jedné skupiny k dispozici více běhů, jedná o speciální nastavení části generátorů na konstantní hodnoty pro všechny iterace. Například pro skupiny longdouble10 je pro kombinaci generátorů RRRRRrrr k dispozici nastavení generátorů RRRRRr1r1r1, RRRRRrrr, R1R1R1R1R1rrr, R2R2R2R2R2rrr a R3R3R3R3R3rrr.
- V rámci některých experimentů chybí některé gpa soubory. Zpracování genetického algoritmu nejspíše nebylo dokončeno v limitu, a tak se některé iterace vůbec neprovedly. Existuje celkem 11193 gpa složek, ve kterých se nachází jen jeden gpa soubor, 3224 se dvěma soubory a 3057221 složek s očekávanými třemi gpa soubory.
- Existuje celkem sedm experimentů, které nemají žádné výsledky a jejich zpracování tak s největší pravděpodobností skončilo chybou nebo vůbec nezačalo. Jedná se například o experiment DDDDDddd skupiny longdouble5.

- Co se týče počtu úspěšně zpracovaných gpa souborů, tj. souborů které mají výsledek před definovaným počtem cyklů, je na tom nejhůře skupina longdouble0. Úplně nejhorší je skupina DDDDDeee, kde bylo úspěšně zpracováno přes 91 % souborů. Ani jedna kombinace generátorů pro skupinu longdouble0 nemá 100 % úspěšně zpracovaných gpa souborů. Ostatní skupiny jsou v tomto ohledu takřka bezproblémové a převážná většina kombinací obsahuje 100 % úspěšně zpracovaných souborů nebo se k tomu alespoň značně blíží.
- Výsledky kombinací EEEEEuuu a DDDDDuuu skupiny singlethread_new nejsou relevantní, neboť mají jen 26 respektive 27 úspěšně zpracovaných gpa souborů.
- Statistické ukazatele prozradily, že skupiny nch a nch_loungdouble obsahují stejná data. S největší pravděpodobností zde došlo k omylu v průběhu předávání dat. To samé platí o skupinách multithred a multithred_new.
- Co se týče mediánu a průměru počtu cyklů za které došlo ke zpracování vychází nejlépe skupina singlethread_old. Naopak na chvostu se umístila skupina longdouble0 následovaná skupinami nch respektive nch_longdouble5. Zajímavé je, že starší verze algoritmu siglethread_old vychází lépe než singlethread_new. Co se týče porovnání vícevláknového zpracování oproti jednovláknovému, nelze jednoznačně označit skupina, u které jednoznačně vychází menší počet cyklů.
- Pokud je předmětem zájmu namísto počtu cyklů doba, po kterou trval výpočet, vychází nejlépe skupina multithread respektive multithread_new. To je logické, neboť zpracování probíhalo paralelně. Naopak nejhůře v tomto ohledu vychází skupina singlethread_new respektive singlethread_old.
- Pro porovnání konzistence jednotlivých experimentů jsou k dispozici percentily 25, 50, 75, 90 a 99 pro dobu zpracování i počet cyklů. Data je možné detailněji prozkoumat v Zeppelin sešitu data_vysledky.
- Pro porovnání některých generátorů nastavených na konstantní hodnoty byla zvolena kombinace generátorů RRRRRddd. Výsledky jsou rozporuplné. například u skupiny longdouble10 vychází u experimentů R1R1R1R1R1ddd, R2R2R2R2R2ddd, R3R3R3R3R3ddd počet cyklů více než dvakrát nižší než u obyčejného RRRRRddd. Naopak u skupiny longdouble5 se toto tak výrazně neprojevilo, a počet cyklů je sice u z části konstantních semínek generátorů menší, ale jen okolo deseti procent.
- Při zkoumání složitosti výstupních rovnic podle počtu levých závorek obsažených v rovnici, vyšla nejhůře skupina singlethread_new s průměrným počtem závorek 17,5.

Naopak o celých 5 závorek méně vyšlo u skupin nch a nch_longdouble5. V porovnání vícevláknových a jednovláknových běhů algoritmu vychází lepe vícevláknový přístup, ale v průměru jen o něco málo více než půl závorky.

- Korelace mezi číslem iterace a počtem cyklů za který algoritmus doběhl odhalila, že tyto proměnné lze považovat za na sobě naprosto nezávislé.

Další zajímavé informace se může uživatel pokusit najít sám ve výsledných Zeppelin sešitech. Ke každému grafu či tabulce je v sešitech uvedeno co zobrazují a jak byla data získána.

ZÁVĚR

V teoretické části této práce byla představena problematika analýzy dat, a to včetně možných problémů při práci s velkými daty. Součástí teoretické části bylo i představení vhodných nástrojů, které by k analýze vzorových dat praktické úlohy mohly být vhodné. Pro praktickou část byl zvolen nástroj Apache Spark, který je v teoretické části porovnáván s alternativním nástrojem Hadoop.

Úvod praktické části je věnován nastavení prostředí a instalaci potřebných nástrojů a technologií. Podařilo se najít takovou kombinaci nástrojů, která umožní projít celým procesem analýzy dat od extrakce dat z archivů přes načtení dat do databáze, analyzování dat pomocí frameworku Spark po vytvoření vizuálních prvků ze získaných výsledků pomocí sešitového rozhraní Zeppelin a grafů z knihovny Bokeh. Tímto nalezeným řešením bylo zadání práce splněno.

Během zpracovávání práce se vyskytlo několik problémů, které se však podařilo vyřešit. Prvním problémem se ukázal být zvolený operační systém Windows, který byl posléze nahrazen operační systémem Linux. Další problém nastal při snaze načítat a zpracovávat data z objemné databázové tabulky, tento problém byl vyřešen načítáním dat po částech pomocí integrovaných funkcí Apache Spark. Poslední problém nastal při prezentaci dat v tabulkové formě integrovaným Zeppelin rozhraním, kdy ne vždy probíhalo řazení správně. Tento problém se podařilo vyřešit pomocí interaktivních Zeppelin prvků kdy v návaznosti na vybrané položce probíhá řazení dat přímo pomocí Sparku.

Výsledné Zeppelin sešity slouží k prezentaci výsledků a byly obohaceny i o části zdrojového kódu, kterým byly získány vizualizované parquet soubory. Díky tomu se čtenář může seznámit se způsobem práce v Spark SQL API.

Nejtěžší částí práce bylo se zorientovat v datech která měla být analyzována. S tímto souvisí i úkol klást správné dotazy tak, aby výsledná data byla relevantní. Potenciál zdrojových dat nebyl nejspíše vyčerpám, ale další bádání už bylo nad schopnosti autora, a to z důvodů časových a také nedostačenou znalostí algoritmů které data vygenerovaly.

Během práce se autor seznámil se světem technologií distributivních výpočtů, práce s velkými daty a nástroji pro analýzu dat. V tomto ohledu byla práce pro autora přínosná a obohacující. Na tuto práci může autor navázat například vyzkoušením Sparku v clusteru namísto lokálního módu nebo třeba vyzkoušet jiný dostupný nástroj, popřípadě najít další informace ve zdrojových datech, které by se daly podrobit hlubší analýze.

POUŽITÁ LITERATURA

[1] REINSEL, David, John GANTZ a John RYDNING. The Digitalization of the World: From Edge to Core [online]. 2018 [cit. 2020-07-05]. Dostupné z: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf

[2] LIANG, Dong, LiZhe WANG, Fang CHEN a HuaDong GUO. 科学大数据与数字地球. Chinese Science Bulletin. 2014, 59(12), 1047-1054. DOI: 10.1360/972013-1054. ISSN 0023-074X. Dostupné také z: http://engine.scichina.com/doi/10.1360/972013-1054

[3] AGGARWAL, Charu C. Data Mining: The Textbook. Cham: Springer International Publishing, 2015. ISBN 978-3-319-14142-8.

[4] Data Mining Tutorial: Process, Techniques, Tools, EXAMPLES. Guru 99 [online]. [cit. 2020-07-06]. Dostupné z: https://www.guru99.com/data-mining-tutorial.html#3

[5] Big data. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2020 [cit. 2020-07-06]. Dostupné z: https://en.wikipedia.org/wiki/Big_data

[6] Scala (programming language). In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2020 [cit. 2020-07-06]. Dostupné z: https://en.wikipedia.org/wiki/Scala_(programming_language)

[7] Sbt Reference Manual. Scala-sbt [online]. [cit. 2020-07-06]. Dostupné z: https://www.scala-sbt.org/1.x/docs/

[8] PUTANO, Ben. A Look At 5 of the Most Popular Programming Languages of 2019. Stackify [online]. 2019 [cit. 2020-07-06]. Dostupné z: https://stackify.com/popular-programming-languages-2018

[9] Python (programming language). In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2020 [cit. 2020-07-06]. Dostupné z: https://en.wikipedia.org/wiki/Python_(programming_language)

[10] Apache Zeppelin. Apache Zeppelin [online]. [cit. 2020-07-06]. Dostupné z: http://zeppelin.apache.org

[11] SCOTT, James. Getting Started with Apache Spark. San Jose: MapR Technologies, 2015.

[12] CHAMBERS, Bill a Matei ZAHARIA. Spark: The Definitive Guide: Big Data Processing Made Simple. Sebastopol: O'Reilly Media, 2018. ISBN 9781491912218.

[13] MariaDB. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2020 [cit. 2020-07-06]. Dostupné z: https://en.wikipedia.org/wiki/MariaDB

[14] Quickstart. Bokeh [online]. Bokeh Contributors, 2019 [cit. 2020-07-06]. Dostupné z: https://docs.bokeh.org/en/latest/docs/user_guide/quickstart.html#userguide-quickstart

[15] Apache Spark. Apache Spark: Lightning-fast unified analytics engine [online]. The Apache Software Foundation, 2018 [cit. 2020-07-06]. Dostupné z: http://spark.apache.org

[16] KALRON, Amir. How do Hadoop and Spark Stack Up? Logz.io [online]. 2020 [cit. 2020-07-06]. Dostupné z: https://logz.io/blog/hadoop-vs-spark

[17] KRIVAA, Karen. Hadoop vs. Spark: Debunking the Myth. GigaSpaces [online]. 2019 [cit. 2020-07-06]. Dostupné z: https://www.gigaspaces.com/blog/hadoop-vs-spark

[18] Spark wins Daytona Gray Sort 100TB Benchmark. Apache Spark: Lightning-fast unified analytics engine [online]. The Apache Software Foundation, 2014 [cit. 2020-07-06]. Dostupné z: https://spark.apache.org/news/spark-wins-daytona-gray-sort-100tb-benchmark.html

[19] NUNNS, James. Spark, Hydra & BigQuery: 5 enterprise alternatives to Hadoop. Computer Business Review [online]. Computer Business Review, 2018 [cit. 2020-07-07]. Dostupné z: https://www.cbronline.com/cloud/spark-hydra-bigquery-5-enterprise-alternatives-to-hadoop-4891737

[20] Spark, Hydra & BigQuery: 5 enterprise alternatives to Hadoop. HDFSTutorial [online]. HDFS Tutorial Team, 2020 [cit. 2020-07-07]. Dostupné z: https://www.hdfstutorial.com/blog/top-hadoop-alternatives

[21] Downloads: Setting up MariaDB Repositories. Https://downloads.mariadb.org/mariadb/repositories [online]. MariaDB Foundation, 2020 [cit. 2020-07-08]. Dostupné z: https://downloads.mariadb.org/mariadb/repositories/#distro=Ubuntu&distro_release=bionic--ubuntu_bionic&mirror=liquidtelecom&version=10.4

[22] MariaDB Package Repository Setup and Usage. MariaDB [online]. MariaDB, 2020 [cit. 2020-07-08]. Dostupné z: https://mariadb.com/kb/en/mariadb-package-repository-setup-and-usage

[23] GUOAN, Xiao. How to Install MariaDB 10.4 on Ubuntu 18.04, Ubuntu 20.04. LinuxBabe [online]. LinuxBabe, 2020 [cit. 2020-07-08]. Dostupné z: https://mariadb.com/kb/en/mariadb-package-repository-setup-and-usage

[24] Download Apache Spark™. Apache Spark: Lightning-fast unified analytics engine [online]. Apache Spark, 2020 [cit. 2020-07-08]. Dostupné z: https://spark.apache.org/downloads.html

[25] Installing on Linux. ANACONDA.DOCUMENTATION [online]. 2020 [cit. 2020-07-12]. Dostupné z: https://docs.anaconda.com/anaconda/install/linux

[26] ANACONDA [online]. Anaconda, 2020 [cit. 2020-07-12]. Dostupné z: https://www.anaconda.com

[27] POLI, Riccardo, W. B. LANGDON, Nicholas F. MCPHEE a John R. KOZA. A field guide to genetic programming. [S.l.: Lulu Press], 2008. ISBN 978-1-4092-0073-4.

[28] JÄGARE, Ulrika. Unified Analytics For Dummies®. Hoboken. John Wiley & Sons, 2019. ISBN 978-1-119-54597-2.

[29] MariaDB Java Client » 2.5.2. MvnRepository [online]. 2020 [cit. 2020-07-29]. Dostupné z: https://mvnrepository.com/artifact/org.mariadb.jdbc/mariadb-java-client/2.5.2

[30] ScalikeJDBC. MvnRepository [online]. 2020 [cit. 2020-07-29]. Dostupné z: https://mvnrepository.com/artifact/org.scalikejdbc/scalikejdbc

[31] CHAN, Alex. Iterating over the entries of a compressed archive (tar.gz) in Scala. Alexwlchan [online]. 2019 [cit. 2020-08-05]. Dostupné z: https://alexwlchan.net/2019/09/unpacking-compressed-archives-in-scala