FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Automatizace testování softwaru na různých verzích platformy Apache Spark
Student:	Karolína Radovská
Vedoucí:	Mgr. Adam Juraszek
Studijní program:	Informatika
Studijní obor:	Znalostní inženýrství
Katedra:	Katedra aplikované matematiky
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Seznamte se s oblastí testování distribuovaného softwaru založeného na frameworku Apache Spark.
V práci se zaměřte na systémové testování, konkrétně na chyby způsobené odlišností různých verzí komponent ekosystému Apache Hadoop, jehož je Apache Spark obvyklou součástí.

Analyzujte problematiku automatického testování SW na různých verzích Apache Hadoop; identifikujte obvyklé problémy.
Zaměřte se na chyby vstupně výstupních formátů, které jsou často způsobeny nekompatibilitou knihoven.
Za tímto účelem navrhněte nástroj sloužící k přípravě prostředí, spouštění testů a jejich vyhodnocení.

V praktické části implementujte podstatnou část tohoto nástroje: přípravu prostředí odpovídající vstupním parametrům (za použití technologií virtualizace), instalaci testovaného SW, definici základních testovacích dat a provedení definovaného testu.
Rozsah implementační části bude upřesněn na základě výsledků analýzy.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Karel Klouda, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 5. října 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Automatizace testování softwaru na různých verzích platformy Apache Spark

Karolina Radovská

Katedra aplikované matematiky
Vedoucí práce: Mgr. Adam Juraszek

16. května 2019

Poděkování

Děkuji za trpělivost a cenné rady vedoucímu práce, Mgr. Adamovi Juraszkovi, a za odborné konzultace Mgr. Zuzaně Vytiskové a Ing. Vladimirovi Emelianovi. V neposlední řadě děkuji Ing. Lucii Renátové za psychickou podporu během psaní této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 16. května 2019 .....

Odkaz na tuto práci

Radovská, Karolina. *Automatizace testování softwaru na různých verzích platformy Apache Spark*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Práce se zabývá automatizací testování distribuovaného softwaru na platformě Apache Spark se zaměřením na chyby vstupně-výstupních formátů způsobené odlišností různých verzí komponent ekosystému Apache Hadoop.

Teoretická část se věnuje ekosystému Apache Hadoop a jeho komponentám s důrazem na Apache Spark a datové formáty.

V praktické části je navržen a implementován nástroj pro automatické testování distribuovaného softwaru založeného na platformě Apache Spark. Implementovaný nástroj automaticky spouští nakonfigurovaný Hadoop cluster a na základě uživatelem dodané konfigurace spouští podmnožinu definovaných testů.

Nástroj je implementován ve skriptovacím jazyce bash. Hadoop cluster je vytvořen pomocí technologie VirtualBox.

Klíčová slova ekosystém Hadoop, Apache Spark, big data testování, automatické testování

Abstract

The thesis deals with the automation of testing of distributed software on Apache Spark with focus on input-output format errors caused by different versions of Apache Hadoop ecosystem components. The theoretical part deals with the Apache Hadoop ecosystem and its components with an emphasis on Apache Spark and data formats. In the practical part a tool for automatic testing of distributed software is designed and implemented, based on Apache Spark platform. The implemented tool automatically runs the configured Hadoop cluster along with a subset of predefined tests based on the user-supplied configuration. The tool is implemented in bash scripting language. Hadoop cluster is created using VirtualBox technology.

Keywords Hadoop ecosystem, Apache Spark, big data testing, test automation

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Big data dnes představují jeden z největších obchodních artiklů v oblasti IT, a počet různých softwarů, které se věnují jejich zpracování, stále narůstá. Spolu s tím narůstá i potřeba tyto softwary efektivně testovat.

Tato práce se zabývá automatizací testování distribuovaného softwaru založeného na frameworku Apache Spark, jednom z nejpopulárnějších nástrojů pro zpracování dat. Práce se zaměřuje na chyby vstupně-výstupních formátů, často zapříčiněné odlišností různých verzí komponent ekosystému Hadoop, s kterými Apache Spark běžně integruje.

Téma jsem si zvolila, neboť existující komerční nástroje pro automatizaci testování big data aplikací se zaměřují především na datovou kvalitu, analýzu a ukládání dat. Tyto nástroje jsou využitelné pro firmy, které se těmito úkony zabývají.

V případě firem dodávajících řešení pro zpracování dat je ale testování softwaru nad různými verzemi komponent ekosystému Hadoop nutnost. Tyto firmy vyvíjejí software bez předchozí znalosti clusteru potenciálního zákazníka, vzniká nutnost sledovat trendy ve zpracování dat. Tím vzniká i potřeba testovat software na různých verzích komponent ekosystému Hadoop a jejich vzájemnou kompatibilitu.

Cílem práce je vytvořit návrh a prototyp testovacího nástroje, který bude tento proces automatizovat, konkrétně provede instalaci Hadoop clusteru obsahující verze komponent dle uživatelovy konfigurace a používající ke zpracování framework Apache Spark, spustí na něm test a vyhodnotí výsledky.

Práce je strukturovaná do čtyř kapitol: Cílem první kapitoly je představit ekosystém Hadoop, popsat architekturu jeho základních komponent a detailně čtenáře seznámit s frameworkem Apache Spark. Druhá kapitola se zabývá testováním. Třetí kapitola obsahuje návrh nástroje pro automatické testování distribuovaného softwaru založeného na frameworku Apache Spark. Čtvrtá kapitola vymezuje rozdíly mezi navrženým a implementovaným nástrojem.

Kapitola 1

Ekosystém Hadoop

Pojem „ekosystém Hadoop“ (nebo krátce Hadoop) je souhrnný název pro více než čtyřicet komponent, které vyvíjí Apache Software Foundation[1] a několikanásobné množství projektů, které s těmito komponentami integrují

Je to platforma, která řeší problémy týkající se zpracování velkých dat. Za velká data jsou považována taková data, jejichž velikost neumožňuje je zachycovat, spravovat a zpracovávat běžně používanými softwarovými prostředky (*3D Data Management: Controlling Data Volume, Velocity, and Variety* [1]). Uváděny jsou tři klíčové vlastnosti, které velká data definují: rozmanitost, rychlost, objem. Rozmanitost dat je dána různou strukturovaností (více v kapitole Formáty velkých dat), rychlost odkazuje na rychlost generování dat, objem na jejich kvantitu.

Pouhé hromadění dat postrádá význam – jejich hodnota tkví v informacích a znalostech, které je z nich majitel schopen získat. Za tímto účelem vznikl ekosystém Hadoop.

Pojmenování ekosystému Hadoop pochází z projektu Apache Hadoop, na jehož základech je ekosystém Hadoop vystavěn.

Po vzoru Apache Hadoopu jsou komponenty ekosystému open source, distribuované, škálovatelné a tolerantní k selhání hardwaru, na němž jsou spouštěny. Ekosystém obsahuje komponenty zaměřené na distribuované úložiště, distribuované výpočty, strojové učení, plánování úloh a další.

Některé z komponent je možné používat i bez Apache Hadoopu. Tyto komponenty zpravidla používají další projekty z ekosystému, aby nahradily funkcionality Apache Hadoopu. (Příkladem je Apache Spark, který lze integrovat např. s Apache Cassandrou, která mu poskytuje úložiště dat, jímž Spark nedisponuje. Obě zmíněné komponenty jsou v podkapitolách popsány.)

O základních komponentách ekosystému Hadoop pojednává podkapitola Apache Hadoop. V dalších podkapitolách jsou popsány komponenty, které základní funkcionalitu Apache Hadoopu rozšiřují.

[1] Kompletní přehled všech komponent, které pro Hadoop ASF vyvíjí, lze nalézt na: https://projects.apache.org/projects.html?category

1.1 Apache Hadoop

Apache Hadoop umožňuje distribuované zpracování velkých dat pomocí clusterů sestavených z nízkonákladových strojů. Je navržen pro škálování od jednotlivých serverů po tisíce strojů, každý z nich poskytuje lokální výpočty a úložiště.

Apache Hadoop nespoléhá na bezchybovost hardwaru, místo toho detekuje selhání strojů na aplikační vrstvě. Nad hardwarem náchylným k selhání tím tvoří vysoce spolehlivou službu. (*Apache Hadoop* [3])

Projekt obsahuje čtyři základní moduly, které budou popsány v následujících podkapitolách: Hadoop Common, HDFS, Hadoop YARN a Hadoop MapReduce.

Tato práce se zabývá Apache Hadoopem 2 (tedy verzemi 2.0 – 2.9).

1.1.1 Hadoop Common

Hadoop Common je považován za jádro frameworku Hadoop. Obsahuje balíček nezbytných knihoven a skriptů. Poskytuje základní služby a procesy ostatním Hadoop modulům.

1.1.2 HDFS

Distribuovaným souborovým systémem (DFS) se rozumí souborový systém, který ukládá data na několika různých počítačích či uzlech clusteru. Umožňuje snadno kontrolovatelný přístup k datům několika klienty. Ti mohou sdílet informace a data stejně, jako by se tomu dělo u lokálního souborového systému.

„Hadoop Distributed File System (HDFS) je distribuovaný souborový systém navržený pro běh na komoditním hardwaru.“ (*HDFS Architecture Guide* [16])

HDFS počítá s tím, že hardware bude nízkonákladový a náchylný k selhání, a proto je architektonicky řešen tak, aby tato selhání odhaloval rychle a byl schopen je automaticky vyřešit. Tím dosahuje vysoké chybové tolerance.

Je určený primárně pro dávkové zpracování (batch processing). Využívají ho aplikace pracující s velkými datesety. Velikost souborů ukládaných na HDFS se typicky pohybuje v řádech gigabajtů až terabajtů, proto HDFS umožňuje zefektivnit práci s těmito soubory přesunutím výpočtů za daty, oproti obvyklému přístupu, kdy jsou za účelem zpracování přesouvána data.

Je založený na Javě.

1.1.2.1 Architektura HDFS

Architektura HDFS je založena na master/slave modelu. Skládá se z jednoho namenodu (master), který spravuje metadata, a mnoha datanodů (slave), kde jsou uložena samotná data.

Hierarchie jmenného prostoru HDFS se podobá hierarchii většiny existujících souborových systémů, umožňuje vytvářet složky a do nich ukládat soubory.

Ty jsou interně rozděleny na bloky dat a uloženy na datanodech.

Replikace dat

V kapitole HDFS bylo uvedeno, že velikost souborů ukládaných na HDFS se pohybuje v řádech terabajtů. Soubory jsou ukládány napříč clusterem, rozdělené na posloupnost bloků stejné velikosti (typicky 128 MB). Bloky jednoho souboru jsou ukládány na různých datanodech.

Aby se zabránilo ztrátě dat při selhání uzlů, jsou bloky replikovány. Informace o replikách spravuje namenode, jejich počet, včetně velikosti bloků, je parametr konfigurovatelný pro každý soubor.

NameNode (master)

Namenode je master server HDFS. Plní úkol koordinační jednotky. Spravuje jmenný prostor, organizuje přístup klientů k souborům, udržuje a spravuje jak datanody, tak datové bloky na nich uložené.

Nenahrává žádná uživatelská data, pouze jejich metadata, jako například umístění bloku, přístupová práva, velikost souboru a podobně. Provádí také operace jako přejmenování, mazání souborů apod.

Zároveň v pravidelných intervalech dostává signály od datanodů, s jejichž pomocí kontroluje stav uzlů a dat na nich uložených, tedy celkové zdraví systému. To mu umožňuje pružně reagovat v případě selhání uzlu – takový datanode označí za odpojený a zahájí replikaci dat na jiný.

DataNodes (slave)

Datanody jsou uzly, ve kterých jsou uložena uživatelská data a na nichž jsou prováděny samotné výpočty.

Provádějí nízkoúrovňové požadavky klientů (čtení, zápis), a periodicky zpravují NameNode o svém stavu a stavu svých dat (ten zjišťují za pomoci kontrolního součtu). Na pokyn namenodu také vytváří, likvidují či replikují bloky dat.

Pomocné uzly

HDFS se může potýkat s mnoha problémy, mezi nejčastějšími například nedostupnost namenode, který zapříčiní nedostupnost celého souborového systému. Těmto situacím lze předcházet ustanovením záložních namenodů. Ty usnadní namenodu práci, v případě jeho nedostupnosti mohou jeho funkci zcela převzít. Jedná se o sekundární namenode, checkpoint namenode a backup namenode.

1.1.3 Apache Hadoop YARN

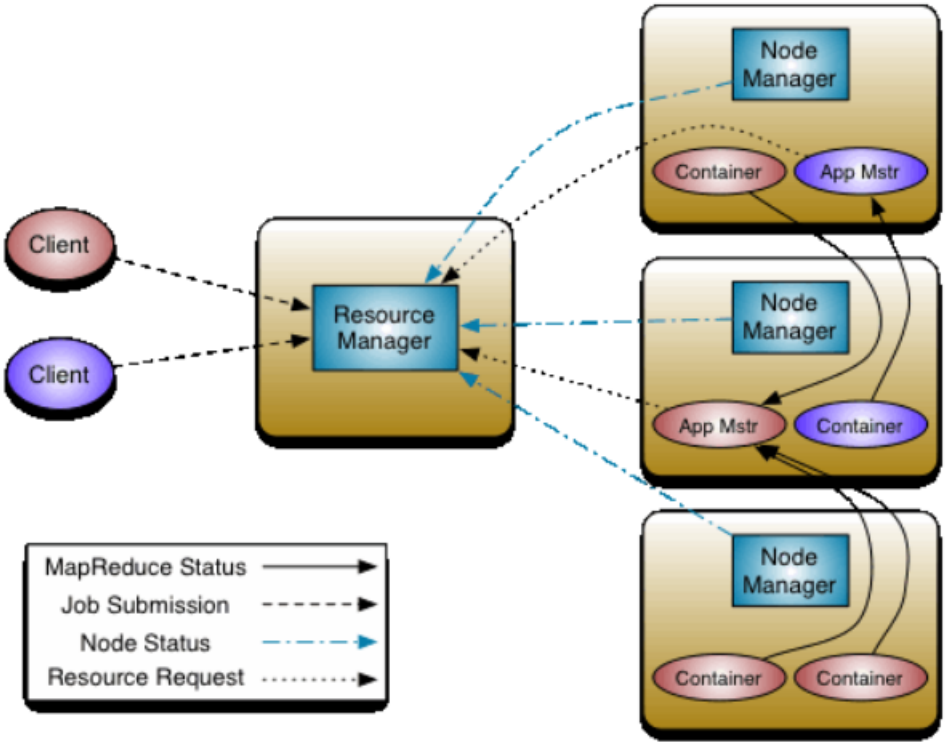
Apache YARN (Yet Another Resource Negotiator) zajišťuje plánování a správu úloh. Vznikl na základě myšlenky oddělit tyto úkony od zpracování dat, které

bylo dříve podporováno pouze pro MapReduce.

Oddělením plánování a správy úloh byly rozšířeny možnosti, jakými lze na Hadoopu zpracovávat data. YARN umožňuje na jednom clusteru lze používat různé frameworky pro různé případy užití.

1.1.3.1 Architektura YARNu

YARN se skládá ze tří komponent – Resource Manageru, Node Manageru a Application Masteru. Ty budou detailněji rozebrány níže.



Obrázek 1.1: Architektura Apache YARNu Zdroj:*Apache Hadoop* [3]

Jak je vidět na obrázku 1.1, Resource Manager je pouze jeden, každý uzel má vlastní Node Manager a každá aplikace má vlastní Application Manager. Aplikací se rozumí úloha [job], nebo DAG (orientovaný acyklický graf) úloh.

Kontejner je balíček fyzických zdrojů (RAM, CPU a další) na každém uzlu.

Resource Manager

Resource Manager spravuje a optimalizuje využití zdrojů clusteru. Rozhoduje o jejich přidělení jednotlivým soupeřícím aplikacím na základě jejich požadavků. Dělí se na dvě komponenty:

- Scheduler, který je odpovědný jen a pouze za alokaci zdrojů (tzv. „pure scheduler“)
- Application Manager, který přijímá úlohy a vyjednává první kontejner pro spuštění jejich Application Masterů. Běh úloh (na rozdíl od scheduleru) kontroluje, v případě selhání úlohu restartuje.

Node Manager

Node Manager pravuje individuální uzel, je zodpovědný za využití jeho zdrojů. Vede si záznamy o jeho zdraví, periodicky o něm zpravuje Resource Manager.

Application Master

Každá aplikace má svůj vlastní Aplication Master. Ten kromě vyjednání zdrojů od Resource Manageru spolupracuje s Node Managerem na spuštění dílčích tasků a jejich průběžné kontrole.

Stejně jako Node Manager zpravuje o svém zdraví Resource Manager, zároveň aktualizuje svoje požadavky na zdroje.

1.1.4 Apache Hadoop MapReduce

MapReduce je programovací model pro souběžné distribuované výpočty nad velkými datasety. Společně s HDFS realizuje myšlenku přenosu výpočtu za daty, což je v případě velkých datových sad efektivnější než opačný přístup (popsáno v kapitole HDFS).

MapReduce pracuje výhradně s páry klíč-hodnota, vstup i výstup jsou typicky uloženy na HDFS. Data jsou před zpracováním rozdělena na nezávislé díly, ty jsou následně zpracovány dílčími úlohami aplikace.

Aplikace (tzv. MapReduce úloh) je soubor konfigurace, vstupních a výstupních lokací dat, funkcí map a reduce.

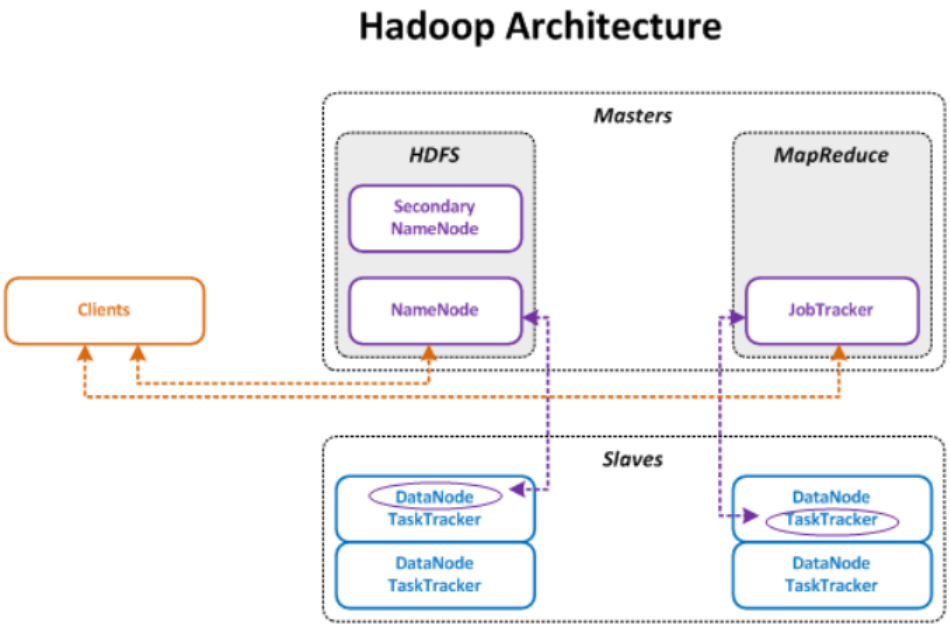
1.1.4.1 MapReduce a HDFS

MapReduce běží nad stejnou množinou uzlů jako HDFS (obrázek Architektura Hadoop). To umožňuje efektivní plánování spouštění úloh na uzlech, kde jsou uložena zpracovávaná data, a tedy výrazné snížení režie a doby běhu programu.

Proces spuštění úloh ovládají dva typy uzlů, které jsou ve vztahu master/slave: JobTracker a TaskTracker.

JobTracker

Masterem je JobTracker, který se stará o plánování úloh s ohledem na to, kde se nacházejí zpracovávaná data. Také úlohy sleduje a v případě jejich selhání je restartuje. Informace o datech získává od namenodu.



Obrázek 1.2: Architektura Hadoop Zdroj:*Hadoop Components* [14]

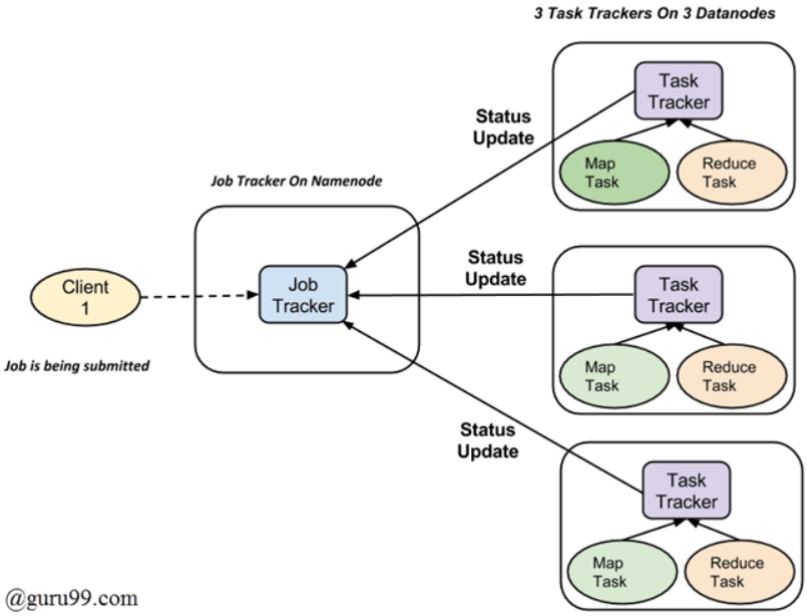
TaskTracker
Slave je TaskTracker, sídlí na každém uzlu a spouští dílčí úlohy zadané JobTrackerem. S tím neustále komunikuje. Pokud komunikace ustane, JobTracker usoudí, že tento uzel selhal a dílčí úlohu přidělí jinému.

1.1.4.2 Životní cyklus MapReduce úlohy

Životní cyklus MapReduce úlohy se skládá z následujících kroků:

1. Úloha je připravena k běhu, zkontrolována a klientem předložena JobTrackeru.
2. JobTracker rozplánuje map fázi (vizte níže) a předá TaskTrackerům.
3. TaskTrackery spustí map funkci. Informují JobTracker o postupu.
4. Jakmile je k dispozici alespoň část výsledků map funkcí, JobTracker rozplánuje reduce fázi (vizte níže) a opět předá úlohu TaskTrackerům.
5. TaskTrackery spustí reduce funkci. Informují JobTracker o postupu.

Fáze Map
Ve fázi Map dochází ke dvěma úkonům:



Obrázek 1.3: Životní cyklus MapReduce úlohy (jobu) Zdroj:*Apache Hadoop MapReduce Concepts* [4]

- Splitting: Dochází k rozdělení vstupních dat na díly (input split). Každý vzniklý díl je předán jako vstup do map funkce a je zpracováván jako samostatná dílčí úloha.

- Mapping: Vstupem do map funkce je jsou díly vstupních dat. Funkce map následně vyprodukuje páry klíč-hodnota.

Fáze Reduce

Ve fázi Reduce dochází ke dvěma úkonům:

- Shuffling sjednotí relevantní výsledky map funkce podle klíče.

- Reducing agreguje výsledky z funkce shuffle a sumarizuje datset.

K tomu, aby začala reduce fáze, nemusí být k dispozici všechny výsledky map funkcí, proto se často obě fáze prolínají. V některých případech není fáze Reduce vyvolána vůbec.

1.2 Apache Spark

Apache Spark je framework pro zpracování dat. Neposkytuje vlastní souborový systém, pracuje v kombinaci s ostatními komponentami ekosystému Hadoop.

Apache Spark byl vyvinut jako alternativa k Apache Hadoop MapReduce, je odezvou na jeho limity a v mnoha ohledech ho překonává.

Průzkumy z roku 2016 (*Apache Spark Market Survey: Cloudera Sponsored Research* [9]) ukazují, že 54 % uživatelů z oblasti zpracování velkých dat používá Apache Spark. Analýzy *Apache Spark Market Forecast 2019-2022* [8] předpovídají růst počtu uživatelů.

Spark překonává Hadoop díky RDD (popsáno v podkapitole Spark Core) v rychlosti v ideálních podmínkách až stokrát, v nejhorších podmínkách minimálně třikrát (Raju [23]), umožňuje oproti dvoufázovému modelu MapReduce iterativní zpracování, podporuje výpočty v paměti, poskytuje zpracování v reálném čase a další funkcionality (blíže rozvedeno v podkapitolách). Další výhodou Apache Sparku je oproti MapReduce větší uživatelská přívětivost: Programátor si může zvolit z několika jazyků.

V následujících podkapitolách jsou popsány komponenty Apache Sparku a architektura Spark aplikace.

1.2.1 Komponenty Sparku

Apache Spark framewrok se skládá z pěti modulů: Spark Core, Spark Streaming, Spark SQL, GraphX a MLlb.

Následující podkapitoly se věnují jejich bližšímu popisu.

1.2.1.1 Spark Core

Spark Core poskytuje veškeré základní funkcionality pro ostatní komponenty Sparku.

Zodpovídá za výpočty v paměti, zotavení se z chyb a rozdělení úloh, je základ pro paralelní a distribuované zpracování velkých datasetů. Obsahuje rozhraní pro programování v Javě, Pythonu, Scale a R.

V tomto modulu jsou implementovány kolekce dat RDD.

Resilient Distributed Dataset

Odolná distribuovaná sada dat (RDD) je základní datová struktura Apache Sparku. Je to neměnná kolekce dat rozdělená mezi jednotlivé uzly clusteru na logické kusy dat (tzv. partition), které umožňují paralelní operace. Neměnnost RDD pomáhá dosáhnout konzistence ve výpočtech.

RDD může vzniknout ze souboru v HDFS (či jiných vhodných zdrojů), nebo transformací již existující RDD. Existují dva typy operací, které lze nad RDD provádět:

- Transformace vracejí jako výsledek další RDD. Patří mezi ně metody map, filter atd.

- Akce vyhodnocují transformace a vracejí výsledek řídícímu programu. Při vyhodnocování je pro každou partition vytvořena dílčí úloha (task). Mezi akce patří metody reduce, collect atd.

Na rozdíl od modelu MapReduce Apache Spark nerozlišuje pouze dvě fáze. Umožňuje řetězení transformací, nehledě na to, zda se jedná o funkce map či reduce. Spark používá odložené vyhodnocení, tedy princip, dle kterého je vyhodnocení výsledků oddáleno na poslední možnou chvíli.

Vyhodnocení trasformací nastává až v okamžiku, kdy je vyvolána akce. Místo mezivýsledků jsou v orientovaném acyklickém grafu závislostí (tzv. lineage graf) ukládány transformace, které mají být nad datasetem provedeny.

Tento graf je zásadní pro zachování chybové tolerance. Zachováním přehledu o transformacích lze transformace znovu aplikovat v případě, že je při selhání uzlu poškozena či ztracena datová sada. RDD se tak vyhýbá nutnosti replikovat data přes několik uzlů, jako je tomu u MapReduce modelu.

1.2.1.2 Spark Streaming

Spark Streaming zajišťuje škálovatelnou proudovou analýzu dat s vysokou propustností a tolerancí k chybě. Funguje na principu micro-batchingu pro zpracování dat v reálném čase: Seskupuje shromážděná data do malých dávek, které předá systému pro dávkové zpracování.

Nabízí také vysokoúrovňovou abstrakci, tzv. DStream (diskretizovaný tok). Interně se jedná o sekvenci RDD, je to kontinuelní tok dat ze souborových systémů, zdrojů, jako je např Apache Kafka (*Apache Kafka Documentation* [5]), nebo získaný vysokoúrovňovými operacemi nad dalšími DStreamy.

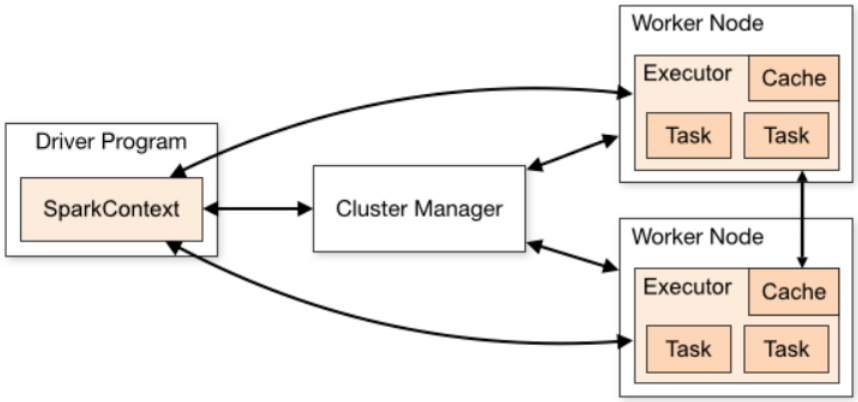
1.2.1.3 Spark SQL

"Spark SQL je modul pro zpracování strukturovaných dat."(*Spark SQL, DataFrames and Datasets Guide* [28])

Oproti základnímu RDD rozhraní provádí dodatečné optimalizace na základě informací o struktuře dat i prováděném výpočtu.

Přistupuje k různým datovým zdrojům, mezi nimi např. Apache Avro (popsáno v kapitole Apache Avro) a Apache Parquet (v kapitole Apache Parquet).

Sparkq SQL odporuje dva módy pro dotazy nad strukturovanými daty: SQL a Dataset API (Shaikh [25]), pro verzi Sparku 2.0 a dále sloučené se zastaralým rozhraním DataFrames.



Obrázek 1.4: Architektura clusteru Zdroj:*Apache Spark Documentation-Cluster Mode Overview* [7]

1.2.1.4 GraphX

GraphX je rozhraní pro práci s grafovými algoritmy. Jelikož je založený na RDD, které jsou neměnné, není vhodný pro použití na grafech, které je třeba měnit.

1.2.1.5 MLlib

MLlib je knihovna pro strojové učení. Poskytuje různé druhy algoritmů pro klasifikaci, regresi, shlukování a další.

Úkolem MLlib je usnadnit praktické strojové učení a učinit ho škálovatelným.

1.2.2 Architektura Sparku

Hlavním řídícím programem Sparku je tzv. driver. Driver obsahuje SparkContext, který koordinuje veškeré běžící procesy a je zodpovědný za převod aplikace do orientovaného acyklického grafu jednotlivých dílčích úloh.

Každá Spark aplikace se tak na clusteru spouští jako sada procesů. SparkContext jí přidělí vlastní exekutor proces, který se nachází na slave uzlu clusteru, provádí na něm paralelně výpočty a ukládá data. Životnost exekutoru je omezena na dobu, po kterou běží dílčí úlohy aplikace.

SparkContext dále zodpovídá i za samotné odeslání kódu aplikace exekutorům a vyjednávání zdrojů u správce clusteru. Ten je na základě požadavků alokuje.

Jednou z výhod Apache Sparku je, že nezáleží na typu správce clusteru, dokud má driver přístup k exekutorům a ty mohou komunikovat mezi sebou. Oficiální dokumentace (*Apache Spark Documentation-Cluster Mode Overview* [7]) uvádí čtyři podporované správce clusterů: Standalone, Apache Mesos, Ha-

doop YARN a Kubernates, který je zatím pouze experimentální a proto nebude dále rozebírán.

1.2.2.1 Standalone

Apache Spark obsahuje vlastního správce clusteru, který se používá při spouštění Sparku ve standalone (tedy samostnatném) módu. Nastavení clusteru v tomto módu je relativně jednoduché, lze je provádět osobně nebo pomocí přiložených skriptů. Vyžaduje instalaci Sparku na každý stroj, který je součástí clusteru.

Tento mód není vhodný pro případy, kdy je cílem kromě Sparku pouštět i jiné aplikace, případně pokud je nutné použít bohatší možnosti rozplánování zdrojů.

1.2.2.2 Apache Mesos

Projekt Apache Mesos se zabývá správou clusterů[2]. Následuje model master/slave.

Pokud je pro správu Spark aplikací zvolen Apache Mesos, používá k vyjednávání zdrojů scheduler a exekutor. Architektura (na obrázku 1.5) se pak skládá ze tří komponent:

- Master nabízí slave uzlům (tzv. agentům) zdroje podle zvolené organizační strategie, vyjednává zdroje pro aplikaci.
- Agent koordinuje zdroje na fyzickém uzlu. Spouští exekutory pro Spark.
- Spark aplikace používá scheduler, který vyjednává zdroje od master uzlu a manipuluje s nabízenými zdroji, a exekutory, které za využití vyjednaných zdrojů spouští na agentech dílčí úlohy.

1.2.2.3 Hadoop YARN

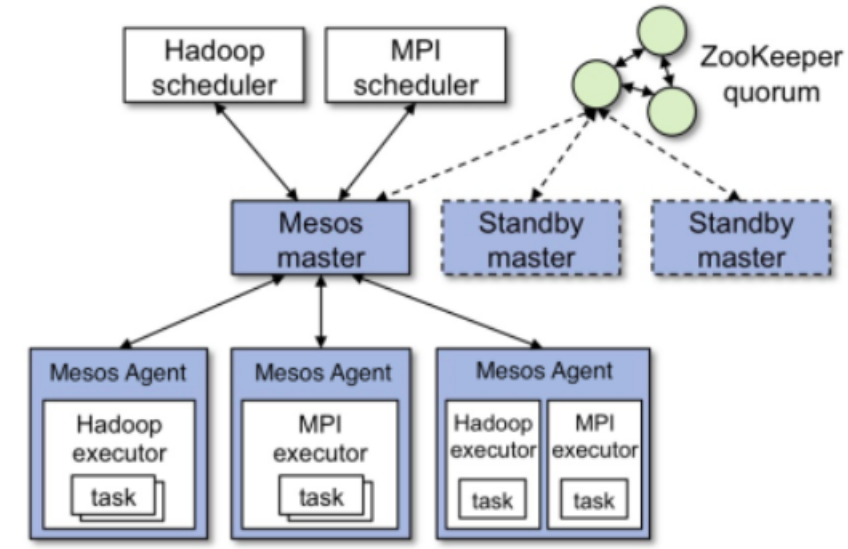
Při běhu na Hadoop YARNu má každá aplikace vlastní Application Master proces. Ten je prvním kontejnerem, který je aplikaci přiděn. Každý exekutor pak běží ve vlastním YARN kontejneru. Oproti modelu MapReduce jsou zdroje pro Spark aplikaci alokovány na celou dobu trvání jejího výpočtu, což výrazně snižuje čas režie.

Podle umístění řídícího programu lze Spark na YARNu pouštět ve dvou módech: yarn-cluster a yarn-client.

Klient mód

Program, který předkládá YARNu aplikaci, se nazývá YARN klient (vizte obrázek 1.1 Architektura YARNu). V tomto módu běží driver na klientovi a

[2]Dokumentace k projektu Apache Mesos je dostupná na https://mesos.apache.org/documentation/latest/index.html



Obrázek 1.5: Architektura Apache Mesos Zdroj:*Mesos Architecture* [20]

není na něj nahlíženo jako na součást YARN clusteru. Není tedy YARNem spravován a jestliže je klient předčasně terminován, ukončí se také Spark aplikace.

Je vhodný pro interaktivní úlohy, kde jsou ihned vidět výsledky (např. pro účely ladění).

Cluster mód

V tomto módu se driver program nachází přímo v Application Masteru, čímž se obchází potřeba aktivního klienta. Proces koordinuje YARN, klient se dotazuje na status Application Mastera a nemusí být přítomen po celou dobu provádění výpočtu. Tento mód je vhodný pro produkci.

1.3 Další Apache projekty

V úvodu kapitoly [ekosystém hadůp] je zmíněno, že Apache Software Foundation obsahuje více než čtyřicet projektů souvisejících se zpracováním (Spark, MapReduce), ukládáním (HDFS) a analýzou velkých dat. Patří mezi ně i projekty, jejichž úkolem je usnadňovat správu clusteru, plánovat spouštění aplikací, zajišťovat bezpečnost, aj.

V následujících podkapitolách jsou popsány jedny z nejčastěji zmiňovaných komponent Hadoop ekosystému.

1.3.1 Apache Pig

Apache Pig je platforma navržená jako abstrakce nad Hadoop MapReduce. Využívá se k rychlé extrakci a anlýze dat, nehledě na to, zda strukturovaných nebo nestrukturovaných.

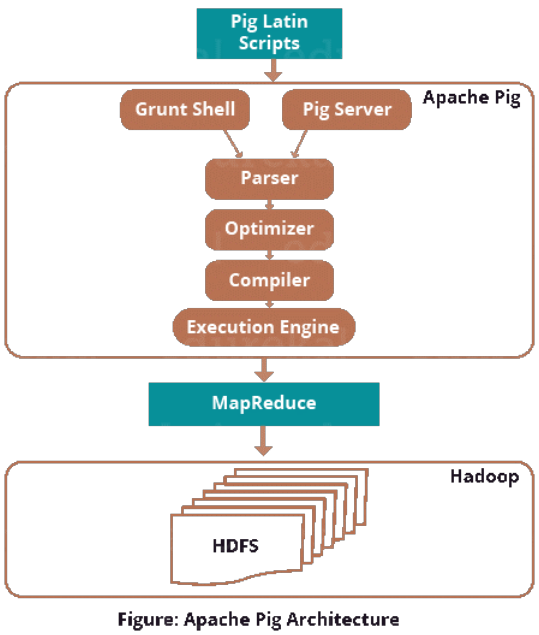
Poskytuje vysokoúrovňový jazyk Pig Latin a běhové prostředí, které skripty psané v tomto jazyce konvertuje na dílčí MapReduce úlohy.

Pig Latin umožňuje realizovat jedním dotazem sekvenci dílčích úloh, čímž až dvacetinásobně snižuje objem kódu (textcitepig) a výrazně usnadňuje dotazování se nad daty. Příkladem takového chování je operátor join.

Dále Pig Latin umožňuje definici vlastních funkcí a jejich vkládání do Pig Latin skriptu a doplňuje MapReduce o vnořené datatypy:

- Uspořádaná n-tice (tuple) – množina polí s různými datovými typy.
- Bag – množina uspořádaných n-tic.
- Mapa – množina párů klíč-hodnota.

Průběh spouštění skriptu ukazuje obrázek 1.6 Architektura Hadoop Pig. Spuštění proběhne pomocí Grunt shellu (poskytuje Apache Pig) nebo na serveru. Parser zkontroluje syntax a předá skript dál v podobě DAGu, kde uzly reprezentují logické operátory a hrany datové toky. Následuje automatická optimalizace a kompilace do série MapReduce úloh.

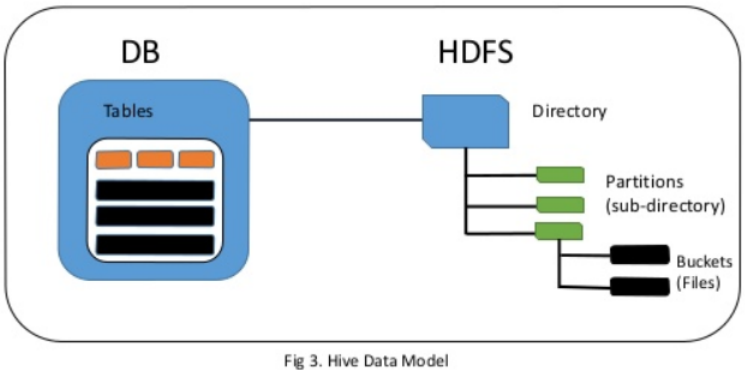


Obrázek 1.6: Architektura Hadoop Pig Zdroj:Sinha [26]

1.3.2 Apache Hive

Apache Hive poskytuje abstrakci nad Apache Hadoopem zaměřenou na skladování strukturovaných a semistrukturovaných dat. Data ukládá ve formátu podobném relačním databázím.

Pro dotazování nad daty poskytuje jazyk HiveQL, který se podobá jazyku SQL. Pracuje se stejnými primitivními datovými typy, obsahuje ale i typy komplexní (mapa, pole, struktura, union). HiveQL umožňuje definici vlastních funkcí.



Obrázek 1.7: Datový model Apache Hive

Datový model Apache Hivu

- Namespace, jehož funkce spočívá v předcházení konfliktů jmen tabulek apod
- Databáze
- Tabulka – má pevné schéma, skládá se z řádků a sloupců. Lze s ní nakládat jako s tabulkou v relační databázi (provádění operací join, union, ...). Může být buď externí (external, ukládající do Hive pouze svá metadata, samotná data jsou v datovém úložišti) nebo interní (managed, kde jsou v Hive data i metadata tabulky).
- Partition – „podadresář“ tabulky, klíč, dle kterého lze sdružovat podobné záznamy, což výrazně usnadňuje dotazování se nad dílčími daty.
- Bucket – do bucketů lze rozdělit každou partition nebo nepartitionovanou tabulku dle hash funkce sloupce v tabulce.

Apache Hive není vhodný pro dotazování v reálném čase. Zpracovává nárazově velké množství dat a zpracování jednoduchých dotazů může trvat řádově i několik minut.

1.3.3 Apache Hbase

Apache HBase je distribuovaná, sloupcově orientovaná, horizontálně škálovatelná NoSQL databáze. Je postavená nad HDFS a využívá jeho chybové tolerance, typicky na něj ukládá data. Oproti HDFS poskytuje nižší latenci pomcí náhodného přístupu k datům. Je vhodná pro analytické zpracování dat.

HBase postrádá koncept neměnných sloupců, je vhodná pro strukturovaná a semistrukturovaná data. Díky horizontálnímu škálování se hodí pro široké tabulky. Data fyzicky ukládá v tzv. regionech. Každý region je obsluhován přesně jedním region serverem.

Architektura Apache HBase následuje master/slave model. Region server v něm zastává roli [slave uzlu], může být odebrán či přidán bez přerušení běhu systému. Region server obsahuje jeden či více regionů, pro něž spravuje požadavky na zápis a čtení, zodpovídá za rozdělení těch regionů, které překročily zadanou maximální velikost.

Master server zodpovídá za administraci a řízení slave serverů. Přiděluje region serverům regiony, stará se o rovnoměrné rozložení zátěže, restartuje servery, které selhaly. Ke koordinaci využívá službu Apache ZooKeeper, bez které není HBase provozuschopná.

Datový model HBase

- Tabulka: skládá se z mnoha řádků
- Řádek: skládá se z klíče řádku [row key] a jednoho nebo více sloupců a jejich hodnot
- Sloupec: skládá se z rodiny sloupce a kvalifikátoru sloupce
- Rodina sloupců (column family): fyzicky spojuje několik sloupců a jejich hodnoty, definuje pro ně vlastnosti jako např. kompresi dat a kódování klíče řádku
- Kvalifikátor sloupce (column quantifier): poskytuje index pro data v rodině sloupce
- Buňka: kombinace řádku, rodiny sloupce a kvalifikátoru sloupce. Obsahuje hodnotu a časové razítko. Může ukládat více verzí své hodnoty.
- Časové razítko (timestamp): přidává se ke každé hodnotě, identifikuje její verzi.

1.3.4 Apache Cassandra

Apache Cassandra je decentralizovaná, sloupcově orientovaná databázová architektura. Využívá modelu peer-to-peer. Všechny uzly jsou strukturně identické a mají stejnou roli. Ačkoliv jsou propojené, jsou na sobě nezávislé. Pokud

selže uzel, na který přišel požadavek, jiný uzel požadavek převezme a obslouží. To v případě Cassandry znamená, že se uzel stane koordinátorem – vyjednává pro klienta data od uzlů, na nichž jsou data uložena. Při selhání uzlu nedojde ke ztrátě dat díky automatické replikaci. (V [31])

Apache Cassandra obshuje následující (pod)komponenty:

- Uzel (node): základní podkomponenta Cassandry, ukládají se na něm data.
- Datové centrum (data center): Jako datové centrum je v modelu Cassandry kategorizována skupina uzlů.
- Cluster: skupina datových center.
- Uzel: Jsou na něm uložena data, provádí obsluhuje požadavky na zápis a čtení dat.
- Commit log: obsahuje záznam o všech operacích zápisu.
- Mem-table: udržuje záznamy v RAM, které se do ní zapisují z Commit logu.
- SSTable: soubor, do nějž se zapisují záznamy z Mem-table, jakmile přesáhnou určitý počet.

Ke komunikaci s datovým modelem Cassandra využívá vlastní dotazovací jazyk: Cassandra Query Language. Klient může přistoupit k libovolnému uzlu, a pomocí cassandra shellu, cqlsh, se na něm dotazovat.

1.3.5 Apache ZooKeeper

Apache Zookeper poskytuje distribuovanou synchronizaci a správu konfiguračních informací. Zajišťuje vzájemnou koordinaci a exkluzi mezi procesy serverů.

S jeho pomocí lze identifikovat uzly v aplikačním clusteru podle jména, bezpečně připojovat a odpojovat uzly, získávat v reálném čase informace o jejich statusech. Dochází též k „zamknutí“ dat, zatímco jsou modifikována. Transakce je tak buď provedena správně, nebo vůbec. Architektura Apache Zookeeperu se skládá z následujících komponent:

- Klient: Uzel v aplikačním clusteru, který přistupuje k informacím serveru.
- Server: Uzel Zookeeperu, který se zaměřuje na obsluhu klienta. Server si s klientem periodicky vyměňují potvrzení o svém aktivním stavu. Pokud server přestane odpovídat, klient se automaticky připojí k jinému.
- Ensemble: Skupina serverů ZooKeeperu

- Vůdčí uzel (leader): Při startu služby je jeden uzel určen jako vůdčí. Je zodpovědný za koordinaci následovníků. Automaticky obnovuje uzly, které selhaly.

- Následovník (follower): Uzel, který následuje instrukce vůdčího uzlu.

1.4 Big data formáty

Velká data (také big data, český ekvivalent se příliš nepoužívá) mohou být strukturovaná, nestrukturovaná nebo semistrukturovaná.

Strukturovaná data jsou vysoce organizovaná, je možné je uložit v jakékoliv relační databázi. Datové typy jsou předem definované, struktura je fixní. Prohledávání dat je možné bez jejich předchozích úprav.

Nestrukturovaná data nemají žádný předem definovaný formát, je obtížné je ukládat i načítat. Příkladem takových dat jsou videa, mp3 soubory aj.

Semistrukturovaná data nelze bez předchozích úprav snadno prohledávat, ani je nelze uložit v relační databázi – je nutná konverze do strukturovaného formátu. Mezi běžně používané semistrukturované formáty patří CSV a Excel.

Semistrukturované formáty lze rozdělit na sloupcově orientované a řádkově orientované, které budou podle zdroje (*An Introduction to Big Data Formats: Understanding Avro, Parquet, and ORC* [2]) rozepsány níže.

Řádkově orientované datové formáty

Zpracování řádkově orientovaných datových formátů probíhá po řádcích. Tyto formáty je vhodné používat, jestliže je nutné přistupovat k většině nebo všem sloupcům v souboru.

Čtení skupiny záznamů např. jednoho konkrétního sloupce je méně efektivní, protože řádkově orientované datové formáty jsou při zpracování děleny opět na množiny řádků. Je tedy nutné přečíst všechny díly zpracovávaného souboru i přesto, že ke zpracování je potřebný jen jeden sloupec.

Mezi řádkově orientované datové formáty patří avro. Avro je formát vyvíjený projektem Apache Avro a patří do ekosystému Hadoop. Podrobněji se mu věnuje kapitola Apache Avro níže.

Sloupcově orientované datové formáty

Sloupcově orientované datové formáty jsou užitečné pro analytické dotazy, které jsou prováděny nad určitou podmnožinou sloupců. Data jsou v těchto formátech nejsou ukládána zleva doprava, ale podle sloupců odshora dolů.

Schopnost čtení pouze relevantních sloupců snižuje čas výpočtu, irelevantní záznamy nejsou vůbec načteny.

Sloupcově orientované formáty používané ekosystémem Hadoop jsou parquet a ORC. Projektu Apache Parquet se věnuje kapitola 1.4.2 Apache Parquet, projektu ORC se věnuje kapitola 1.4.3 Apache ORC.

1.4.1 Apache Avro

Apache Avro je nástroj k serializaci dat. Z dat vytváří řádkově orientovaný binární formát, který je možné komprimovat a rozdělit na určitý počet dílů. To z něj činí vhodný formát vstupních dat pro souběžné zpracování.

Vzniklo jako alternativa k na javě založenému serializačnímu API, kterým disponuje Apache Hadoop. Jeho hlavní výhoda spočívá v tom, že je jazykově neutrální (podporuje jazyky Java, C++, python, Ruby, ...). Podporuje statické i dynamické datové typy.

Klíčovou vlastností formátu avro je přidání schématu souboru do souboru s daty. Popisují následující: typ souboru, umístění záznamu, jméno záznamu a pole v záznamu spolu s jejich korespondujícími datovými typy.

Schémata jsou definována v json formátu, jsou ukládána přímo v souboru s daty a s jejich pomocí lze serializované hodnoty (bez metadat) v binárním formátu ukládat úsporněji.

Využití Avra probíhá ve čtyřech krocích:

1. Navržení a tvorba schématu (podle dat)
2. Načtení schématu do programu; to lze udělat buď pomocí parsers knihovny (přímé čtení schématu), nebo kompilací schématu pomocí avra (vygeneruje class file, která koresponduje se schématem)
3. Serializace dat pomocí serializačního api
4. Deserializace dat pomocí deserializačního api

Avro se typicky používá při write-heavy operacích.

1.4.2 Apache Parquet

"Apache Parquet je sloupcový úložný formát dostupný pro jakýkoli projekt v ekosystému Hadoop, nehledě na volbu frameworku pro zpracování dat, datového modelu nebo programovacího jazyka."(*Apache Parquet* [6])

Apache Parquet formát je specificky navržený k tomu, aby odděloval metadat od dat. To umožňuje rozdělení sloupců do mnoha parquet souborů (record-shredding a assembly algoritmus) a specifikace schémat komprese na úrovni sloupců.

Hierarchie souboru je následující:

- Stránka (page) – z hlediska komprese a kódování nedělitelná jednotka.
- Kus sloupce (column chunk) – obsahuje alespoň jednu stránku.
- Skupina řádků (row group) – dělí horizontálně data do řádků. Obsahuje přesně jeden kus každého sloupce z datasetu. Je sloupcově organizovaná, což umožňuje souboru těžit z benefitů sloupcově orientovaných formátů.
- Soubor s metadaty – skládá se z několika skupin řádků.

1.4.3 Apache ORC

Datový formát Apache ORC (Optimalized Row Columnar) vznikl za účelem zrychlení zpracování souborů a redukce jejich velikosti. Obsahuje stejné datové typy jako Apache Hive (byl pro něj vyvinut).

Je optimalizován pro sloupcové čtení, ale obsahuje i podporu pro rychlé nalezení podstatných řádků.

Díky indexování umožňuje pokročilou kompresi dat. Dělá data do tzv. pruhů (stripes). Pruhy jsou na sobě nezávislé. Pro každý sloupec v pruhu je udržována statistika obsahující informace jako minimální a maximální hodnotu, výskyt prázdných hodnot apod. Díky tomu dotaz při zpracování přeskakuje na pruh, který je pro něj relevantní.

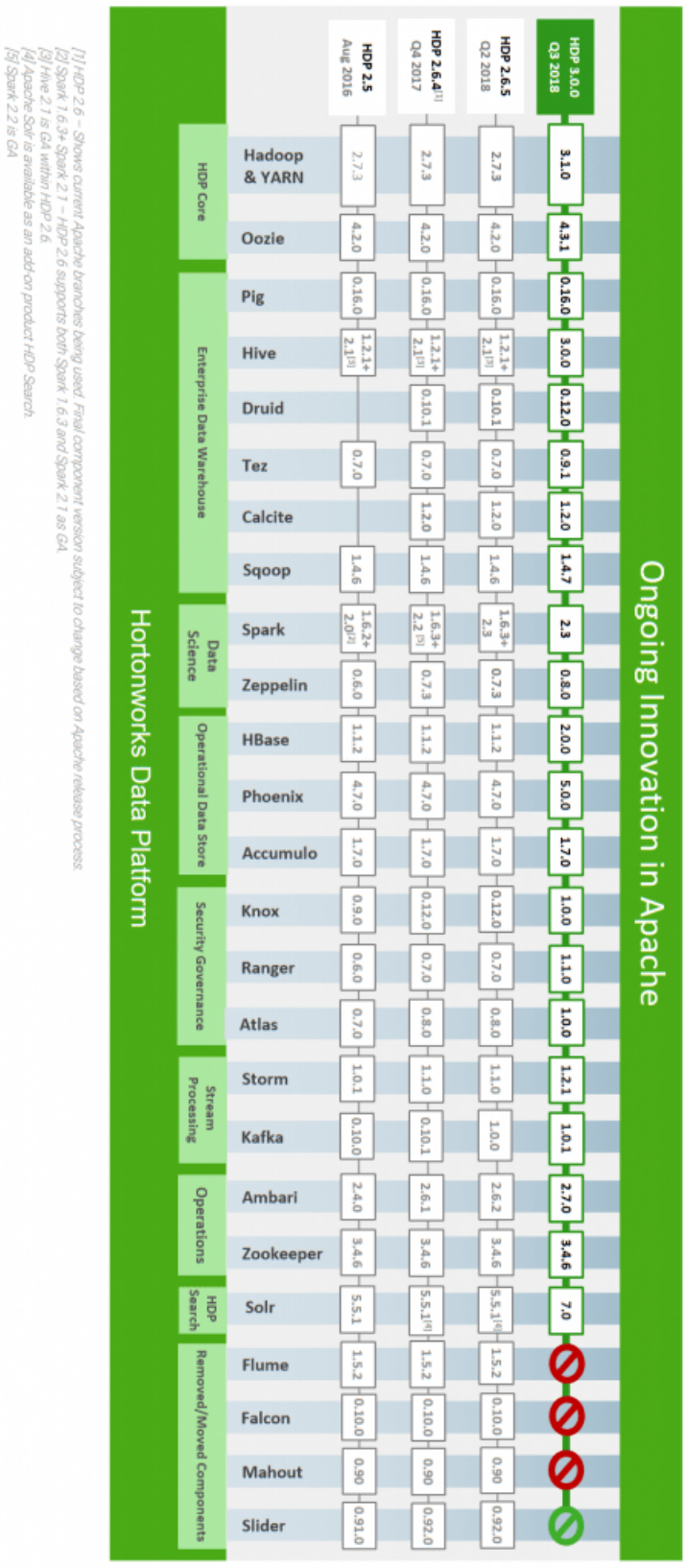
1.5 Hadoop distribuce

Distribuce Hadoopu vznikly díky potřebě usnadnit práci s instalací Apache projektů. Apache Hadoop je poskytován pouze v podobě tar souborů, vůbec se nezabývá např. závislostmi mezi projekty. Projektů je mnoho a vyvíjeny jsou samostatně, což činí správu systému obtížnou.

Distribuce poskytují již otestovaný balíček komponent a tím i funkční Hadoop instalaci. Často také poskytují podporu. (*Hadoop Illuminated: Hadoop Distributions* [15]) Na obrázku 1.8 Verzování komponet v distribuci HDP jsou vidět kombinace komponent, které používá distribuce Hortonworks Data Platform (HDP) do verze 3.0.

Hortonworks je druhá nejpoužívanější distribuce Hadoopu. Jako nejpoužívanější distribuce je hodnocena Cloudera Hadoop Distribution (CDH).(Marr [19], *Top 6 Hadoop Vendors Providing Big Data Solutions in Open Data Platform* [30], Butler [12], *The Top 5 Hadoop Distributions* [29]) CDH i HDP poskytují nástroj pro snadnou instalaci a správu clusteru s webovým rozhraním. Obě distribuce jsou si funkcionalitami velmi podobné. V roce 2018 došlo k oznámení sloučení obou distribucí.

Spolu s CDH a HDP byly v pětici nejlepších distribucí v roce 2016 Forresterem vyhlášeny distribuce MapR, IBM a Pivotal.



22

Obrázek 1.8: Verzování komponent v distribuci HDP Zdroj: *Ongoing Innovation in Apache* [21]

Kapitola 2

Testování softwaru

Testování softwaru je pojem různými autory definovaný různě, například ve vztahu k hledání chyb a zahrnutí či vyloučení statického testování (revizí dokumentace a kódu) z této definice. Zpravidla je považováno za dílčí součást řízení kvality softwaru, zaměřenou na výstupy jednotlivých vývojových procesů – dokumentaci, kód či spustitelný produkt, nikoliv na kvalitu těchto procesů a jejich schopnost produkovat kvalitní kód.

Dle ISTQB (*ISTQB Glossary* [17]) jde o proces zahrnující všechny aktivity životního cyklu softwaru (statické i dynamické) zaměřené na plánování, přípravu a evaluaci softwaru a souvisejících výstupů za účely určení míry naplnění požadavků, demonstrace jejich použitelnosti pro daný účel a nalezání defektů.

Práce bude dále používat následující definici Havlíčkové a Roudenského (Roudenský a Havlíčková [24]): Jde o proces řízeného spouštění softwarového produktu s cílem zjistit, zda splňuje specifikované či implicitní potřeby uživatelů. Testování zkoumá softwarový produkt, čímž získává informace o jeho kvalitě, chápané jako stupeň naplnění požadavků a přání uživatelů. Jeho obsahem je především sběr a analýza informací; nalezené defekty jsou vedlejším produktem této činnosti.

Software nelze otestovat tak, aby byl zcela bez chyb, a ani to není cílem testování. Lze se pokrytím okrajových případů a případným omezením podporovaných prostředí tomuto stavu přiblížit, ale obvykle nejde o vhodné využití zdrojů. Ke zlepšení finální kvality produktu, o kterém víme, že nebude bezchybný, se používají opatření v rámci řízení rizik, nastavení priorit testování na stěžejní funkcionality, ale i uživatelská dokumentace a seznamy známých chyb.

Kvalitu softwaru lze měřit (či odhadnout) na základě ukazatelů, jakými jsou množství nalezených chyb (či chyb vyšší závažnosti pro použitelnost softwaru), množství otestovaných požadavků či případů použití z celkového počtu, pomocí poměru nalezených a uzavřených defektů a dalších. Tyto ukazatele pak slouží pro hodnocení softwaru, úpravu vývojových procesů pro snížení chybo-

vosti produktu a rozhodování.

Testování obsahuje řadu dalších procesů, než je samotné provádění. Dle ISTQB jsou základní testovací procesy tyto:

1. Plánování a řízení
2. Analýzu a návrh
3. Implementace a provádění
4. Vyhodnocení výstupních kritérií a reporting
5. Činnosti ukončení testů

Tato práce je primárně zaměřena na implementaci a provádění, ostatní fáze budou dále zmíněny spíše okrajově.

Některé zdroje (např. *ČSN ISO/IEC/IEEE 29119-Softwarové a systémové inženýrství - Testování softwaru* [13]) rozlišují pojmy chyba, defekt a selhání, tato práce je však používá zaměnitelně, případně jejich lokální kontext dále rozvádí při použití.

2.1 Způsoby testování

Tato kapitola se věnuje způsobům testování, které jsou relevantní pro implementaci a bude na ně v dalších kapitolách navázano:

- whitebox, blackbox a greybox testování
- konfirmační a regresní
- funkční a nefunkční
- automatické

Existuje řada dalších způsobů testování a jejich rozdělení, například manuální a poloautomatické jako protipól již zmíněného automatického testování (rozdělení podle vykonavatele testu), průzkumné a formalizované testování (rozdělení podle existence/využití testovací dokumentace) a další.

Všechny způsoby testování popsané v této kapitole je pak možno rozdělit na pozitivní a negativní (také označované jako testy splněním a testy selháním). Pozitivní testy ověřují, že testovaný funkční či nefunkční požadavek byl splněn a odpovídá specifikaci, negativní testy ověřují selhání v případě, kdy požadavek očekává neúspěch, neboť nebyly naplněny podmínky pro úspěšné provedení (například selhání funkcionality pro hodnoty mimo platný rozsah).

2.1.1 Whitebox, blackbox a greybox testování

Podle testerovy znalosti vnitřní struktury testované oblasti softwaru se hovoří o whitebox testování pro případy, kdy je pro testera dostupný kód a s ním znalost dosažitelných průchodů, stavů a vnitřních komponent, blackbox pro případy, kdy implementace známa není, a greybox pro jejich kombinaci.

2.1.1.1 Blackbox testing

Podle ISTQB (*ISTQB Glossary* [17]) je black box testování buď funční nebo nefunkční testování bez reference na interní strukturu komponenty nebo systému.

Tester poskytuje systému vstupy a následné výstupy porovnává s očekávanými výsledky, aniž by mu byla známa struktura či implementace kódu. Proto se tato metoda používá zejména u systémového a akceptačního testování (blíže rozvedeno v odpovídajících kapitolách 2.2.3 a 2.2.4).

Platí, že čím vyšší úroveň testování (tedy čím větší „skříňka“ je testována), tím užitečnější tato metoda je. Příkladem chyb, které může odhalit, jsou chyby v rozhraní, špatné či chybějící funkce nebo chyby v datových strukturách.

Aby bylo blackbox testování efektivní, vyžaduje specifikace softwaru (bez nich je obtížné navrhovat testovací případy). Mezi jeho metody dle (Patton [22]) patří:

- Třídy ekvivalence: Testovací případy jsou vytvářeny pro skupiny vstupních dat, od kterých jsou očekávány stejné výsledky. Tím se sníží množství vstupů, kterým se musí tester zabývat.
- Analýza hraničních hodnot: Jsou identifikovány hraniční hodnoty ve vstupech a výstupech. Pro každou hranici jsou napsány dva testovací případy (z obou stran hranice).
- Testování přechodu mezi stavy: Funkční chování softwaru je převedeno do stavového automatu. Následně se vytváří testovací případ pro každý stav automatu, každý přechod automatu, nebo pro každou posloupnost stavů (podle požadavků na důkladnost testování).
- Rozhodovací tabulka: Testovací případy obsahují všechny kombinace vstupních hodnot a očekávaných výstupů. Nejlépe využitelné v případech, kdy různé kombinace vstupů vedou k rozdílným výstupům.
- Testování případů užití: Testovací případy jsou navrhovány podle očekávaného chování dle specifikace či uživatelské dokumentace.

2.1.1.2 Whitebox testing

Podle ISTQB je white box testování založeno na analýze interní struktury komponenty nebo systému.

Testování je prováděno mimo uživatelské rozhraní, testerovi je známa struktura a implementace komponenty či systému, sám určuje korektnost výstupů. Whitebox metoda mimo jiné umožňuje testerovi analýzou kódu odhalit jeho nežádoucí části.

2.1.1.3 Greybox testing

Jedná se o kombinaci whitebox a blackbox testování. Aplikuje se především u integračního testování – tester má přístup k algoritmům a datovým strukturám, na jejichž základě navrhuje testovací případy, ale samotné testování probíhá na uživatelské úrovni.

2.1.2 Konfirmační a regresní testování

Konfirmační a regresní testování jsou příklady testování probíhajícího opakovaně. Pojmy spolu úzce souvisí a test může být zařazen jako jeden či druhý způsob pouze na základě kontextu, se kterým je spuštěn.

2.1.2.1 Konfirmační testování

Konfirmační testování (také zvané retestování) je opětovné testování případů, ve kterých byly v minulosti nalezeny chyby, za účelem ověření opravené funcionality. (Bureš et al. [11])

2.1.2.2 Regresní testování

Regresní testování ověřuje, že změny v kódu nezapříčinily chyby v se změnami nesouvisejících oblastech. V ideálním případě by mělo být prováděno při opravení chyb, přidání nových funkcionalit či úpravě kódu dle požadavků, proto je vhodná jeho automatizace.

2.1.3 Funkční a nefunkční testování

Způsoby testování lze rozdělit na funkční a nefunkční na základě požadavků a vlastností softwaru, které mají testy pokrývat.

2.1.3.1 Funkční testování

Funkční testování testuje funkční požadavky. Porovnává to, co systém dělá, oproti tomu, co by dělat měl, tedy má za úkol odhalit nejen chyby ve zdokumentovaných a požadovaných funkcích, ale i odhalit funkce nezdokumentované či nechtěné.

Testování probíhá podle testovacích případů vystavěných na základě případů užití, které jakkoli souvisejí s požadovanými funkcemi softwaru.

Testování probíhá stejným způsobem, jakým by systém používal koncový uživatel.

2.1.3.2 Nefunkční testování

Nefunkční testování netestuje specifické funkce softwaru, ale jeho vlastnosti, to znamená např. spolehlivost, bezpečnost, stabilitu. Měří charakteristiky systému, zjišťuje, jak se systém chová navenek.

Spadají sem testy výkonu, zátěžové testy, stres testy a testy použitelnosti, spolehlivosti, bezpečnosti, podpory a další.

2.1.4 Automatické testování

Automatickým testováním se (podle *ISTQB Glossary* [17]) rozumí „použití softwaru pro provádění nebo podporu testovacích činností, např. řízení testů, návrh testů, provádění testů a kontrola výsledků“. Je přínosem především pro velké projekty, kde je podstatnou náplní práce testera repetitivní činnost, která zvyšuje riziko lidské chyby. Příkladem je regresní testování již zmíněné v podkapitole 2.1.2.2.

Aspekty, které v takových případech staví automatické testování nad testování manuální, jsou dle Patton [22]:

- Rychlost
- Výkonnost
- Přesnost
- Redukce zdrojů
- Simulace prostředí
- Neúnavnost

Dále mohou již vytvořené testy probíhat nezávisle na přítomnosti testera v noci či mimo pracovní dobu a nezatěžovat tak systémy v době většího vytížení. Automatizace umožňuje provedení testů, které by byly manuálně nerealizovatelné (např. zasílání vysokého počtu požadavků v krátkém časovém úseku), a lze se spolehnout na jejich shodnost.

I přes zmíněné přínosy není vhodné automatizovat veškeré testy. Ne vždy existuje náhrada za manuální testování a lidský faktor při vyhodnocování správnosti. Nevýhodou automatického testování je slepota vůči chybám mimo jeho validace – nezachyceným výjimkám, chybám grafického rozhraní apod.

Vhodné prostředí a nástroje pro automatizacimohou být finančně nákladné a v případě nevhodně zvolených testů pro automatizaci může časová nákladnost udržování testů, jejich definice, návrh a vytvoření převážit čas ušetřený jejich prováděním.

Mezi nejčastěji automatizované testy patří jednotkové, zátěžové, integrační, penetrační a end-to-end testy. S vyšší prioritou bývají automatizovány testy ověřující funkce nejdůležitějších součástí systému (nejčastěji využívaných, funkcí,

ze kterých plyne zisk, nebo prerekvizit pro vysoký podíl jiných funkcí) a testy často chybových či komplikovaných funkcí.

Pro automatizaci jsou nevhodné zejména nově navržené testy, které dosud nebyly spuštěny manuálně, testy funkcionalit, jejichž požadavky se často mění, a testy, které jsou spouštěny jen pro konkrétní případy. Do této kategorie spadají i případy, kdy testování vyžaduje lidský úsudek. Automatizace šetří více zdrojů pro zralejší software, s méně se měnícím rozhraním, kódem a funkcionalitami se snižuje čas potřebný pro aktualizaci a opravy testů.

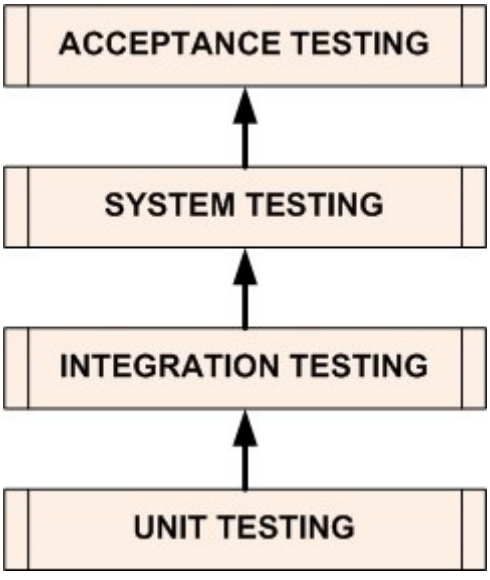
Automatickým testováním distribuovaných systémů se dále bude hlouběji zabývat ještě podkapitola .

2.2 Úrovně testování

Patton [22] dělí testování v rámci vývoje na jednotkové, integrační a systémové testování. U zákazníka následně probíhá testování akceptační. Toto rozdělení vychází z v-modelu, který mapuje fáze testování na fáze vývoje ve vodopádovém modelu.

Představují hierarchické rozdělení od testů nejmenších součástí po celý software. Hlavním kladem tohoto přístupu je omezení množství ještě nenalezených chyb ve fázi, kdy se již jedná o větší část testovaného systému a původ chyby nemusí být na první pohled zřejmý.

Způsoby testování popsané v kapitole výše se podle svojí povahy mohou vztahovat na všechny tyto úrovně testování nebo jen některé.



Obrázek 2.1: Úrovně testování Zdroj:*Software Testing Levels* [27]

2.2.1 Jednotkové testování

Jednotkové testování (také označované jako testování komponent) je pohledu vývojového cyklu nejnižší úroveň testování a zpravidla je provádí vývojář za účelem odhalení chyb už při vytváření kódu.

Jedná se o testování individuálních hardwarových či softwarových komponent (*ISTQB Glossary* [17]), kde komponentou se rozumí minimální část systému, která může být otestována samostatně. V procedurálním programování se může jednat například o funkci nebo individuální program, v objektově orientovaném programování o metodu. Důležitá je právě atomičnost jednotkového testování, existují ale i přístupy, které za jednotku považují samostatný modul aplikace. Jelikož se obvykle dá takový model rozdělit na více otestovatelných jednotek, tento přístup se nedoporučuje.

Tato metoda spadá do kategorie whitebox testování, a používá k dosažení svého cíle různé nástroje, drivery, falešné objekty a podobně.

2.2.2 Integrační testování

Existují dva druhy integračního testování (*ISTQB Glossary* [17]):

- Integrační testování komponent odhaluje chyby v rozhraní a komunikaci komponent.
- Integrační testování systémů kontroluje interakci systémů a jejich kombinace.

Při integračním testování jsou jednotky zkombinovány a otestovány jako skupina. Účelem je odhalit chyby v jejich interakci. Podle toho, zda za jednotku považujeme komponentu či systém, se může jednat o whitebox, greybox či blackbox testování, a podle toho je také provádí buď vývojář, nebo tester.

K integračnímu testování lze přistupovat různě – čtyři základní princpy jsou:

- Big Bang – dojde ke zkombinování a otestování většiny nebo všech jednotek najednou; od systémového testování se tento přístup liší tím, že integrační testování testuje pouze interakci jednotek, ne celý systém, jak je tomu u systémového testování.
- Top Down – používá se při Top Down vývoji, tedy v případě, že se prvně vyvíjejí vysokoúrovňové jednotky a postupuje se k nízkoúrovňovým. Stejným způsobem probíhá testování, které musí simulovat dosud nenaprogramované nízkoúrovňové jednotky.
- Bottom Up – používá se při Bottom Up vývoji (analogicky jako u Top Down metody). V tomto případě jsou simulovány dosud nenaprogramované vysokoúrovňové jednotky.

- Sendvičová (hybridní) metoda – kombinuje přístupy Top Down a Bottom Up.

Integrační testování je záhodno automatizovat, zvlášť při Top Down a Bottom Up přístupu, neboť se musí provádět pokaždé, když je integrována nová jednotka.

2.2.3 Systémové testování

Systémové testování ověřuje, že integrovaný systém splňuje předepsané požadavky (*ISTQB Glossary* [17]). Jedná se o nejvyšší úroveň testování prováděného v rámci vývoje softwaru. Testován je hotový produkt (Patton [22]), a to tzv. end-to-end metodikou – kontrola funkcionalit aplikace probíhá ve scénářích, v jakých by je použil zákazník a to včetně pořadí, v jakém by byly použity.

Hodnotí se specifikované funkční i nefunkční požadavky, které by měl systém splňovat. V této kategorie testů se používají funkční a nefunkční testy, regresní i konfirmační testy.

Jedná se o blackbox testování, provádí je tester.

2.2.4 Akceptační testování

Podle *ISTQB Glossary* [17] se jedná o formální testování podle uživatelových potřeb a požadavků, které ověřuje, zda je systém pro uživatele (či zákazníka) akceptovatelný, tedy že produkt splňuje očekávané cíle a má určitou kvalitu. Častou podmínkou pro převzetí softwaru zákazníkem bývá splnění konkrétních metrik: software neobsahuje žádné kritické chyby, doba odezvy systému nepřekračuje určitý čas, počet nalezených problémů nekritických pro jeho použití k danému účelu nepřekračuje stanovený počet a dalších.

Jedná se o blackbox testování. Akceptační testování může probíhat na straně zákazníka (*ISTQB Glossary* [17]), k omezenému či kompletnímu akceptačnímu testování ale může docházet i na straně prodávajícího před samotným předáním, aby byly případné defekty v kritických funkcionalitách bez újmy na obchodních vztazích.

2.3 Testování distribuovaného softwaru založeného na frameworku Apache Hadoop

Testování distribuovaného softwaru založeného na frameworku Apache Hadoop, nazývané „big data“ testování, typicky zahrnuje testování infrastruktury, databází a funkčních požadavků. Důležitým prvkem je testování architektury. Špatně navržený distribuovaný systém vede k problémům s výkonem, přičemž rychlost zpracování je jedním ze stěžejních požadavků pro práci s velkými daty.

V této podkapitole je testování big data aplikací popsáno se zaměřením na systémové testování, zabývá se tedy funkčními i nefunkčními požadavky, jak již bylo řečeno v kapitole 2.2.3 Systémové testování.

Nefunkční testování big data aplikací klade největší důraz na testy výkonu (*Big Data Testing – Complete beginner's guide for Software Testers* [10]). Měří především načítání a propustnost dat, rychlost zpracování a výkon komponent a nalézá tak výkonnostní slabá místa celého procesu. Například u načítání dat je výkon ovlivněn zejména vhodným formátem dat ve kterém jsou data uložena a celkově je výsledek silně ovlivněn použitým hardwarem, klíčové jsou tedy nejen dosažené absolutní výsledky, ale i výsledky při škálování – funkce nárůstu času při zvětšeném objemu dat a vliv přidání dalších uzlů do použitého clusteru na rychlost zpracování.

Dalšími podstatnými nefunkčními testy jsou pro big data aplikace testy převzetí služeb při selhání (tzv. „failover“ testy). Failover testy ověřují, že selhání některého z uzlů clusteru nenaruší hladký běh aplikace, nemá vliv na korektnost výsledků a nezpůsobí ztrátu dat.

Pro testy spolehlivosti či bezpečnosti se big data aplikace od klasických liší jen minimálně, s výjimkou testů uživatelské přívětivosti, kde jsou požadavky na použitelnost aplikace neškolenou osobou výrazně sníženy povahou Apache Hadoopu.

Funkční testování big data aplikací má podobné cíle jako u softwaru určeného pro běh na jednom zařízení (zmíněné v kapitole 2.1.3 Funkční a nefunkční testování). Uživatelské rozhraní big data aplikace je testováno na základě uživatelských požadavků. Zvláštní důraz je kladen na validaci dat již v okamžiku jejich zadání či před vstupem do HDFS, ověřováno je například schéma souborů. (Justus Selwyn [18])

V podkapitolách jsou dále prodobněji rozebrány fáze testování big data testování, testování kompatibility různých verzí komponent ekosystému Hadoop a automatizace big data testování.

2.3.1 Fáze testování

Velká data, které big data aplikace zpracovávají, mohou být strukturovaná, semistrukturovaná či nestrukturovaná (jak bylo popsáno v kapitole 1.4 Big Data formáty). Pocházet mohou z různých zdrojů – sociální sítě, média, logy a podobně. Proto je před testováním samotné aplikace nutný test datové kvality, které ověří konzistenci, přesnost a další charakteristiky dat. Tím je zaručeno, že chyby nalezené při testování zapříčinila aplikace a nikoliv vadná data.

Data jsou následně předána ke zpracování – je potřeba ověřit jejich přesun na HDFS (či jiné úložiště). Následuje zpracování dat, jehož výstup je opět nutné validovat.

Následuje přenos zpracovaných dat do datového skladu a jeho kontrola. V rámci big data testingu je nutné otestovat i analýzu, která je nad získanými daty prováděna, včetně její vizuální reprezentace.

Celý proces testování se dělí na tři fáze:

1. Ověření platnosti dat: Tzv. „pre-Hadoop“ fáze, validace vstupních dat z externích zdrojů (sociální sítě, média, logy...) a jejich korektního přesunu do HDFS či jiného big data zdroje.

2. Validace zpracování klade nejvyšší důraz na validaci dat po zpracování.

3. Validace výstupu: Kontrola korektního přesunu výstupních dat z big data zdrojů do cílového systému a jejich interpretace.

Ověření platnosti dat a validace výstupu probíhá mimo Hadoop cluster, proto je v podkapitole Validace zpracování níže detailněji rozebrána pouze validace zpracování, na kterou je zaměřena tato práce.

2.3.1.1 Validace zpracování

Na začátku této fáze se na korektní lokaci v HDFS (či jiném Hadoop úložišti) nachází data, která jsou připravena ke zpracování. Základní principy validace zpracování jsou analogické pro všechny tři způsoby zpracování, které Apache Hadoop umožňuje: dávkové, interaktivní a v reálném čase.

Tato práce se zabývá zpracováním dat na Apache Sparku. Apache Spark zpracovává data dávkově i pro „streaming“ – realizuje streaming pomocí micro-batchingu (již zmíněného v kapitole 1.2.1.2 Spark Streaming. Přístup k testování je proto stejný jako pro dávkové zpracování.

Příchozí data jsou shromažďována, dokud není splněna podmínka, která spouští jejich další zpracování – např. množství nových dat či uplynulý časový interval od poslední dávky.

Po zpracování je nutné vystavit kontrole také výsledná data (tzv. „post-hadoop“ validace). Testuje se správné umístění dat na korektní lokaci, jejich správná podoba, obsah.

2.3.2 Kompatibilita různých verzí komponent ekosystému Hadoop

Projekty spadající do ekosystému Hadoop jsou vyvíjeny samostatně. To negativně ovlivňuje kompatibilitu komponent. Na základě této myšlenky vznikly distribuce (kapitola 1.5 Hadoop distribuce), jejichž účelem je poskytnout balíček ověřeně vzájemně dobře fungujících verzí komponent.

Existují společnosti, dodávající software bez předchozí znalosti zákazníkova prostředí a konkrétní konfigurace (a to jak z pohledu použitých komponent a jejich verzí, tak jejich konkrétní konfigurace). Pro zaručení kompatibility by v ideálním případě bylo třeba, aby testování probíhalo na všech kombinacích a to nejen mezi komponentami, ale také mezi verzemi. Časová náročnost a celková obtížnost takového testování by výrazně převýšila jeho užitek, vybírá se tedy na základě priorit a plánovaného využití jen podmnožina komponent

a častým problémem tedy bývá potřeba použít neočekávanou (a tedy netestovanou) kombinaci, která do testovacího plánu zahrnuta nebyla, či konfrontace s chybami integrace ze strany samotných komponent.

Z těchto důvodů existuje silný tlak na alespoň částečnou automatizaci testování.

2.3.2.1 Chyby vstupně-výstupních formátů

Za chyby vstupně-výstupních formátů jsou v této práci považovány chyby způsobené čtením z nebo zápisem do jiného než výchozího formátu.

Ekosystém Apache Hadoop zpracovává strukturovaná, semi-strukturovaná a nestrukturovaná data. Za účelem efektivního zpracování a ukládání dat také definuje vlastní datové formáty, o nichž se zmiňuje podkapitola 1.4 Big data formáty. To zvyšuje pravděpodobnost chyby a tudíž potřebu důkladného testování.

2.3.3 Automatizace

Automatizace by ideálně měla být provedena pro všechny fáze testování distribuovaných systémů, nicméně pro samotný processing je výrazně složitější, než pro otestování správného uložení či načtení dat z HDFS nebo jiného big data zdroje. Neexistuje pro ně zatím žádný standard ani metodika, která by byla obecně použitelná.

Z pohledu úrovně testování by bylo možno vytvořit automatizované jednotkové, integrační i systémové testy. Systémové testování by pak bylo nejvíce využito v případě, jež nastala naposledy s hromadným přechodem od použití MapReduce k používáním Sparku, kdy bylo třeba ověřit správné chování pro jiný engine, ale kompatibilita komponent mezi sebou nebyla ovlivněna.

K automatizaci testování distribuovaných systémů již vznikly nástroje - např. Querysurge nebo Testingwhiz, ty se ovšem zaměřují zejména na validaci výstupů a spíše databázové testování, nikoliv však na problémy kompatibility a další. (Toto zaměření ovšem odpovídá využití distribuovaných systémů některými společnostmi čistě jako úložiště pro data přicházející či vycházející z analytických nebo data management procesů.)

Kapitola 3

Návrh testovacího frameworku

V této kapitole je popsán návrh testovacího frameworku pro testování vstupně-výstupních formátů na různých verzích komponent ekosystému Hadoop.

Nástroj se zaměřuje na systémové testování dávkového zpracování dat pomocí Apache Sparku. Zabývá se pouze druhou fází big data testování, jak je popsána v kapitole [validace procesu], tedy validací procesu. Neposkytuje dodatečné testy (testy výkonu, failover testy).

Navržený testovací nástroj dodává tříuzlový cluster s jedním namenodem a dvěma datanody, realizovaný pomocí technologie VirtualBox. Pevný počet uzlů byl zvolen z toho důvodu, že změna počtu uzlů nemá na chyby vstupně-výstupních formátů vliv. Pro testování byla vybrána distribuce Apache Hadoop, protože na rozdíl od jiných distribucí [odkaz:zmíněných v kapitole xyz] umožňuje konfiguraci clusteru pomocí příkazové řádky, nevyžaduje interakci s webovým rozhraním.

Kromě Apache Sparku a Apache Hadoopu, které jsou pro tento nástroj zásadní, byly pro otestování vstupně-výstupních formátů vybrány následující komponenty: Apache Hive, Apache Hbase, Apache Avro, Apache Parquet a Apache ORC.

Následující podkapitoly se věnují popisu struktury testovacího nástroje a jeho funkcionality.

3.1 Struktura testovacího nástroje

Tato podkpitola obsahuje popis složek a souborů, které testovací nástroj obsahuje, a vysvětluje, k čemu je používá. Strukturu zobrazuje obr. 3.1 Struktura testovacího nástroje.

3.1.1 Uživatelská konfigurace

Uživatel interaguje s následujícími položkami testovacího nástroje:

Složka test_tool/cluster_metadata
Složka obsahuje informace o clusteru, podle nichž uživatel provede nastavení svého softwaru tak, aby byl s clusterem kompatibilní.

Složka test_tool/documentation
Složka obsahuje dokumentaci. V podsložce metadata_definition jsou popsána metadata ke vstupním souborům jednotlivých testovacích případů. Podsložka tc_definition popisuje testovací případy. Dále složka obsahuje soubor readme, který funguje jako dokumentace k testovacímu nástroji.

Složka test_tool/test_results
Do složky jsou po skončení testovacího procesu zapsány výsledky.

Složka test_tool/tested_sw
Do složky uživatel vkládá svůj kód. V podsložce user_sw je uložen testovaný software. V podsložce test_cases se nachází skripty s realizovanými testovacími případy. Soubor run_sw.sh obsluhuje instalaci uživatelova softwaru.
Skripty dodává uživatel.

Složka test_tool/user_configuration
Složka obsahuje soubor default.conf. Jedná se o vzor, do nějž uživatel vyplní vlastní požadavky na konfiguraci clusteru. Souborů může uživatel vytvořit ve složce více. Nástroj provede postupně konfiguraci podle všech souborů tak, jak je popsáno v kapitole [rekonfigurace cluseru].
V tomto souboru také uživatel definuje, které testovací případy chce nad deklarovanou sadou komponent spustit.

Soubor test_tool/run_tool.sh
Skript run_tool.sh obstarává běh testovacího nástroje. Je spuštěn až po dokončení veškeré nutné konfigurace ze strany uživatele. Spouští veškeré ostatní skripty.

3.1.2 Interní složky a soubory

Všechny materiály, které nevyžadují zásah uživatele a zároveň nejsou potřebné pro uživatele z hlediska nastavení testovaného softwaru, jsou umístěny ve složce test_tool/internal.
Její podsložky jsou následující:

Podsložka data
Složka data obsahuje dvě podsložky: input a expected. Podsložka input obsahuje vstupní data pro testovací případy. Podsložka expected obsahuje očekávané výstupy testovacích případů.

Podsložka scripts
Tato složka obsahuje skripty potřebné k obsluze nástroje.

- start_cluster.sh – spouští cluster
- configure_cluster.sh – konfiguruje cluster podle souborů v test_tool/configuration/*.conf
- run_tests.sh – kopíruje data na HDFS, spouští skript run_tc.sh
- run_tc.sh – spouští jeden testovací případ, porovnává výstup testu s očekávaným výstupem, zapisuje výsledky
- remove_configuration.sh – odstraňuje konfiguraci clusteru
- stop_cluster.sh – vypíná cluster

Podsložka vm
Obsahem podsložky vm jsou tři virtuální stroje tvořící cluster.

Podsložka other
Tato podsložka obsahuje soubory související s konfigurací Hadoop clusteru. Obsahuje dvě podsložky: versions a xmls.

Podsložka versions
Podsložka versions obsahuje soubory pojmenované po komponentách Hadoop ekosystému. Každý soubor obsahuje seznam verzí komponenty. Ke každé verzi je uveden hyperlink, na němž je dotyčná verze dostupná ke stažení.

Podložka xmls
Podložka xmls obsahuje předkonfigurované xml soubory pro konfiguraci Apache Hadoop clusteru. Korespondují se soubory ve složce test_tool/cluster_metadata.

3.2 Funkcionalita testovacího nástroje

Funkcionalitu testovacího nástroje lze rozdělit na tři části – přípravu prostředí, spuštění testů a vyhodnocení výsledků. Protože má testovaný nástroj umět spustit více testovacích cyklů, patří do funkcionality nástroje i smazání prostředí a jeho opětovná konfigurace (hlouběji zmíněná v kapitole 3.2.4 Rekonfigurace clusteru).

3.2.1 Příprava prostředí

Přípravou prostředí se rozumí automatická konfigurace a spuštění clusteru s uživatelem zadanými parametry.

3.2.1.1 Uživatelem zadávané parametry

Parametry zadává uživatel do souborů s příponou .conf ve složce test_tool/user_configuration. Jako vzor slouží uživateli poskytovaný soubor default.conf na téže lokaci.

Soubor default.conf obsahuje seznam komponent a seznam či rozsah testovacích případů ke spuštění. Ke každé komponentě uvede uživatel její požadovanou verzi. Není-li uvedena žádná verze, komponenta není nainstalována. Neplatí pro základní komponenty Apache Spark a Apache Hadoop.

Není-li specifikován počet ani rozsah testovacích případů, jsou spuštěny všechny.

3.2.1.2 Start a konfigurace clusteru

Skript test_tool.sh spustí pomocí skriptu start_cluster.sh virtuální stroje. Následně skript test_tool.sh spustí pomocí skriptu configure_cluster.sh konfiguraci Apache Hadoop clusteru. Je-li ve složce test_tool/configuration více než jeden soubor s příponou .conf, vybírá abecedně soubor, jehož konfigurace ještě neproběhla. Pro každou komponentu v konfiguračním souboru je stažena pomocí hyperlinku uvedeném v souboru test_tool/internal/other/versions/component_name příslužná verze. Poté jsou na spuštěný cluster zkopírovány složky test_tool/internal a tested_sw. Na clusteru jsou komponenty Hadoop nakonfigurovány s využitím proměnných prostředí a předem připravených xml souborů (složka test_tool/other/xmls). Konfigurační skript v této fázi nekontroluje, zda jsou uvedené verze vzájemně kompatibilní, jedná se o zodpovědnost uživatele.

3.2.2 Spuštění testů

Spouštění testů je realizováno skriptem run_tests.sh. Tento skript nahraje vstupní data na příslušnou lokaci v clusteru, která je uváděna v definici testovacího případu, a nainstaluje uživatelem dodaný software.

Skript run_test.sh po přesunu dat vytvoří tabulku výsledků result_table a spustí run_tc.sh pro každý testovací případ z uživatelem zadaného rozsahu testovacích případů. Skript run_tc.sh přijímá jako parametr číslo testovacího případu, který spouští. Data jsou zapsána na opět podle definice testovacího případu.

3.2.3 Vyhodnocení výsledků

Skript run_tc.sh provede porovnání očekávaných výstupů s korespondujícími výstupy testovacího případu, jehož číslo dostal jako parametr. Výsledek zapíše jako nový řádek do tabulky výsledků result_table.

Tabulka result_table má dva sloupce: TC a RESULT. TC reprezentuje číslo testovacího případu. RESULT reprezentuje výsledek (ilustrováno v tabulce výsledků xy).

Nástroj rozeznává tři výsledky: porovnání souborů skončilo úspěchem (success), výstup chybí nebo se liší od očekávaného výsledku (fail) a chyba při běhu testovacího případu (error).

Po spuštění posledního testovacího případu je tabulka je pod názvem konfiguračního souboru, jehož specifikace byla použita pro konfiguraci clusteru, uložena do složky results.

Poté jsou smazány veškeré použité soubory a složky.

TC	**Result**
0001	SUCCESS
0002	FAIL
0011	ERROR

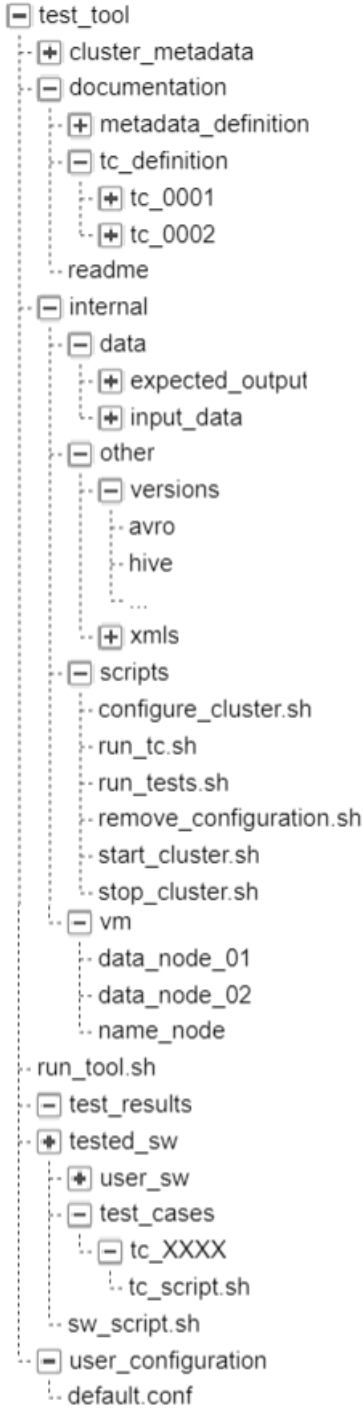
Tabulka 3.1: Tabulka výsledků testů

3.2.4 Rekonfigurace clusteru

Rekonfigurace clusteru je podmíněna úspěšnou likvidací předchozí konfigurace clusteru (jedná se především o nakonfigurované soubory, dočasné složky vytvořené pro běh komponet Hadoop ekosystému, proměnné prostředí).

Konfiguračnímu souboru, jehož definice verzí komponent byla použita k poslední konfigurací clusteru, je odebrána přípona .conf.

Následuje spuštění nového testovacího procesu tak, jak je popsán v kapitole Příprava prostředí níže.



Obrázek 3.1: Struktura testovacího nástroje

Kapitola 4

Implementace

Implementační část práce si klade za cíl vytvořit prototyp automatického testovacího nástroje, který je navržen v kapitole 3.

V této kapitole jsou uvedeny poznámky k implementaci. Testovací nástroj je obsažen v příloze včetně dokumentace. Na základě dohody s vedoucím práce je v příloze namísto clusteru z virtuálních strojů poskytnut návod na jeho konfiguraci.

Následující kapitoly popisují rozdíly v implementaci, kterým oproti návrhu došlo, a jejich důvody.

4.1 Příprava prostředí

Příprava prostředí pro spouštění definovaných testů se zabývá automatickou konfigurací a automatickým spouštěním Hadoop clusteru.

Distribuce s webovým rozhraním pro instalaci a správu clusteru zmiňované v kapitole (distribuce) jsou k automatické instalaci nevhodné. Proto byla pro automatickou instalaci Hadoop clusteru zvolena distribuce Apache Hadoop, která je dodávána ve formě tar souborů.

Na základě zkušeností nabytých při automatizaci procesu lze říci, že zakomponování automatizace konfigurace clusteru do návrhu testovacího nástroje bylo chybným krokem.

Automatizace přípravy prostředí probíhala nejdříve na jednom virtuálním stroji v pseudomódu (tedy v testovacím módu pro Apache Hadoop, kdy je spouštění na různých strojích simulováno spouštěním úloh na různých JVM procesech), následně na trojuzlovém clusteru z virtuálních strojů. V obou případech pomocí proměnných prostředí (JAVA_HOME, HADOOP_HOME, atd.) a předpřipravených soubrorů xml s konfigurací základních komponent Apache Hadoop a Apache Spark.

Při tomto procesu byly odhaleny následující problémy konfigurace:

- Chyby neposkytují dostatečné informace o svém původu.

- Chyby nejsou vždy deterministické. (Např. kód, spuštěný na nakonfigurovaném clusteru dvakrát po sobě, bez zjevného důvodu při druhém spuštění selže.)

- Nečekané chyby v rozsahu, který není možné rozumně ošetřit.

Z hlediska testovacího nástroje je dalším problémem nemožnost uživateli zabránit v instalaci nekompatibilních verzí komponent. V návrhu je sice uvedeno, že se jedná o zodpovědnost uživatele, nicméně jak se zmiňuje kapitola [verze ah komponent v testování], projekty vyvíjejí komponenty nezávisle a jako takové vyžadují důkladné otestování. Toto testování provádějí pouze distribuce nad určitou podmnožinou komponent a jejich verzí.

S využitím nabytých znalostí se autorka kloní k závěru, že nelze předpokládat dostatečnou hloubku uživatelových znalostí o závislostech mezi verzemi projektů na to, aby byl schopen nástroj využít. To činí testovací nástroj prakticky nepoužitelným.

Od automatické konfigurace Hadoop clusteru bylo na základě těchto zkušeností a po konzultaci s odborníkem (ing. Vladimirem Emelianovem, Big Data Practice Leadem společnosti Ataccama Software.sro) upuštěno. Po domluvě s vedoucím práce byl návrh testovacího nástroje pro implementaci částečně přepracován, konkrétně ve fázi přípravy prostředí.

Zodpovědnost za konfiguraci Hadoop clusteru byla přesunuta na uživatele. Uživatel obdrží cluster virtuálních strojů, na němž nakonfiguruje vlastní verzi libovolné distribuce.

Nástroj se pak stará o automatické spouštění testů na tomto Hadoop clusteru.

Nevýhodou této změny je pro uživatele nutnost angažovat se v přípravě běhového prostředí. Dále to znamená zvýšené náklady na paměť – předpokládá se opětovné využití již nakonfigurovaného clusteru, virtuální stroje je třeba někde ukládat. Opak by znamenal nutnost provádět konfiguraci prostředí při každém běhu nástroje, což přímo popírá základní princip automatizace (tj. šetření času testera).

Výhodou je získaná flexibilita ve výběru použitých distribucí.

4.2 Spouštění testovacích případů

Spouštění testů je realizováno stejným způsobem, jak o něm hovoří návrh. Na cluster jsou pomocí protokolu ssh zaslány definované testy a následně spuštěny.

Z důvodů uvedených v kapitole (porovnání výsledků) byla do implementace přidána možnost v rámci jednoho testovacího případu více testovacích skriptů.

4.3 Porovnání výsledků

Při porovnávání výsledků testovcích případů bylo nutno vyřešit problematiku porovnávání formátů avro, parquet a ORC.

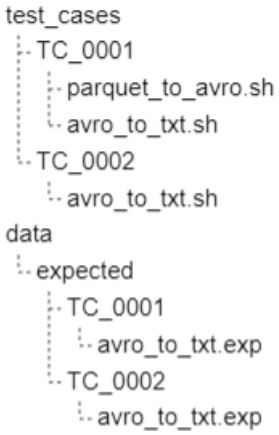
Způsob řešení byl zvážen dvojí:

- Porovnání hashe
- Konverze souboru do snadno porovnatelného formátu (např. txt)

Porovnání hashem je nevhodné, protože odlišné verze knihoven sestavují schémata souborů odlišně. To vede k nahlášení domnělé chyby kvůli rozdílům v hashi i přesto, že odlišná verze knihovny je schopna soubor přečíst. V reakci na toto chování by bylo nutné buď přegenerovat očekávané výsledky podle uživatelových knihoven, nebo manuálně retestovat všechny testy, které skončily chybou.

Zvolen byl přístup konverze souboru. Konverzi lze realizovat pomocí dodatečné definice testovacích případů, které ověřují zápis do obtížně porovnatelných formátů. Uživatel na základě definice dodá dva skripty: první skript, který realizuje čtení z libovolného formátu zapisující do obtížně porovnatelného formátu, a druhý skript, realizující čtení ze souboru vytvořeného prvním skriptem a zápis do txt či jiného snadno porovnatelného formátu.

Oba skripty jsou spouštěny v rámci stejného testovacího případu. Aby se zamezilo nejasnosti o původu chyby při selhání testu, bude proveden doplňkový test čtení z obtížně porovnatelného formátu.



Obrázek 4.1: Porovnání výsledků testu zápisu do formátu avro

Závěr

Cílem práce bylo navrhnout nástroj pro automatizované testování distribuovaného softwaru založeného na frameworku Apache Spark a implementovat podstatnou část tohoto nástroje jakožto prototyp. Uživateli mělo být umožněno definovat verze komponent Hadoop ekosystému, a úkolem nástroje bylo provést automatickou instalaci Hadoop clusteru podle uživatelových požadavků. Následně měl nástroj na vytvořeném clusteru provést definovaný test nad uživalem dodaným softwarem a vyhodnotit výsledky.

Během práce bylo zjištěno, že automatizace instalace Hadoop clusteru je nevhodná, protože nelze zaručit kompatibilitu verzí uživatelem zadaných komponent a tudíž funkčnost nakonfigurovaného Hadoop clusteru. Tento dílčí cíl nebyl splněn.

Implementace se proto v tomto směru odpovídajícím způsobem odchýlila od návrhu. Implementovaný prototyp testovacího nástroje spouští definované testy a vyhodnocuje jejich výsledky na Hadoop clusteru dodaném uživatelem. Hlavní cíle práce byly splněny.

Práci by bylo možné rozšířit sepsáním plné sady testovacích případů pro datové formáty specifické pro Apache Hadoop, prozkoumáním možností automatické konfigurace clusteru pro distribuce používající k instalaci clusteru webové rozhraní, nebo implementací přívětivějšího uživatelského rozhraní pro testovací nástroj.

Literatura

[1] *3D Data Management: Controlling Data Volume, Velocity, and Variety*. [cit. 2019-02-14]. MetaGroup. URL: https://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf.

[2] *An Introduction to Big Data Formats: Understanding Avro, Parquet, and ORC*. [cit. 2019-04-02]. Nexla. 2018. URL: https://thinksis.com/wp-content/uploads/2018/10/Nexla_Whitepaper_Introduction-to-Big-Data-Formats-Saket-Saurabh.pdf.

[3] *Apache Hadoop*. [cit. 2019-02-07]. Apache Hadoop. URL: https://hadoop.apache.org.

[4] *Apache Hadoop MapReduce Concepts*. [cit. 2019-03-17]. MarkLogic. URL: https://docs.marklogic.com/guide/mapreduce/hadoop#id_82011.

[5] *Apache Kafka Documentation*. [cit. 2019-03-02]. Apache Kafka. URL: https://kafka.apache.org/documentation/.

[6] *Apache Parquet*. [cit. 2019-03-18]. Apache Parquet. URL: https://parquet.apache.org/.

[7] *Apache Spark Documentation-Cluster Mode Overview*. [cit. 2019-02-18]. Apache Spark. URL: https://spark.apache.org/docs/latest/cluster-overview.html.

[8] *Apache Spark Market Forecast 2019-2022*. [cit. 2019-02-17]. Market Research Media Ltd. 9. dub. 2018. URL: https://www.marketanalysis.com/?p=159.

[9] *Apache Spark Market Survey: Cloudera Sponsored Research*. [cit. 2019-02-17]. Taneja Group. URL: http://tanejagroup.com/profiles-reports/request/apache-spark-market-survey-cloudera-sponsored-research#.WCCdPC0rK70.

[10] *Big Data Testing – Complete beginner's guide for Software Testers*. [cit. 2019-04-26]. ISTQB Try QA. URL: http://tryqa.com/big-data-testing/.

[11] M. Bureš et al. *Efektivní testování softwaru: klíčové otázky pro efektivitu testovacího procesu*. Grada Publishing, a.s., 2016. ISBN: 9788024755946. URL: https://books.google.cz/books?id=VXHLDQAAQBAJ.

[12] Brandon Butler. *The top 5 Hadoop distributions according to Forrester*. [cit. 2019-02-20]. 20. led. 2017. URL: https://www.networkworld.com/article/3024812/the-top-5-hadoop-distributions-according-to-forrester.html.

[13] *ČSN ISO/IEC/IEEE 29119-Softwarové a systémové inženýrství - Testování softwaru*. Český normalizační institut. 2015.

[14] *Hadoop Components*. [cit. 2019-04-07]. URL: https://sites.google.com/site/amitsciscozone/home/hadoop/hadoop-componentsg.

[15] *Hadoop Illuminated: Hadoop Distributions*. [cit. 2019-03-13]. Hadoop Illuminated. URL: https://hadoopilluminated.com/hadoop_illuminated/Distributions.html.

[16] *HDFS Architecture Guide*. [cit. 2019-02-07]. Apache Hadoop. URL: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[17] *ISTQB Glossary*. [cit. 2019-02-21]. ISTQB. URL: http://glossary.istqb.org/.

[18] Nachiyappan .S a Justus Selwyn. "PRE HADOOP AND POST HADOOP VALIDATIONS FOR BIG DATA". In: *International Journal of Mechanical Engineering and Technology (IJMET)* 8 (10 říj. 2017). [cit. 2015-10-31]. ISSN: 0976-6359. URL: http://www.iaeme.com/MasterAdmin/UploadFolder/IJMET_08_10_066/IJMET_08_10_066.pdf.

[19] Bernard Marr. *Big Data: Who Are The Best Vendors In 2017*. [cit. 2019-02-20]. 30. led. 2017. URL: https://www.linkedin.com/pulse/big-data-who-best-hadoop-vendors-2017-bernard-marr.

[20] *Mesos Architecture*. [cit. 2019-03-17]. Apache Mesos. URL: http://mesos.apache.org/documentation/latest/architecture/.

[21] *Ongoing Innovation in Apache*. [cit. 2019-04-13]. URL: https://2xbbhjxc6wk3v21p62t8n4d4-wpengine.netdna-ssl.com/wp-content/uploads/2018/07/asparagus.png.

[22] R. Patton. *Software Testing*. Sams, 2006. ISBN: 9780672327988. URL: https://books.google.cz/books?id=MTEiAQAAIAAJ.

[23] Vinod Raju. *Is There Still A Future For MapReduce?* [cit. 2019-02-27]. Jigsaw Academy. 11. 30 2018. URL: https://analyticstraining.com/still-future-mapreduce/.

[24] P. Roudenský a A. Havlíčková. *Řízení kvality softwaru*. Computer Press, Albatros Media a.s., 2017. ISBN: 9788025145197. URL: https://books.google.cz/books?id=xRvqCwAAQBAJ.

[25] Ty Shaikh. *Batch Processing-Apache Spark*. [cit. 2019-03-10]. K2 Data Science a Engineering. 24. led. 2019. URL: https://blog.k2datascience.com/batch-processing-apache-spark-a67016008167.

[26] Shubham Sinha. *Pig Tutorial: Apache Pig Architecture And Twitter Case Study*. [cit. 2019-03-20]. Edureka. 26. ún. 2019. URL: https://www.edureka.co/blog/pig-tutorial/.

[27] *Software Testing Levels*. [cit. 2019-03-16]. Software Testing Fundamentals. URL: http://softwaretestingfundamentals.com/software-testing-levels/.

[28] *Spark SQL, DataFrames and Datasets Guide*. [cit. 2019-03-10]. Apache Spark. URL: https://spark.apache.org/docs/latest/sql-programming-guide.html.

[29] *The Top 5 Hadoop Distributions*. [cit. 2019-02-20. Big Data Analytics News. 8. ún. 2016. URL: https://bigdataanalyticsnews.com/the-top-5-hadoop-distributions/.

[30] *Top 6 Hadoop Vendors Providing Big Data Solutions in Open Data Platform*. [cit. 2019-05-16]. IntelliPaat. 16. květ. 2019. URL: https://intellipaat.com/blog/top-6-hadoop-vendors-providing-big-data-solutions-in-open-data-platform/.

[31] Manoj V. "Comparative Study of NoSQL Document, Column Store Databases and Evaluation of Cassandra". In: *International Journal of Database Management Systems* 6 (srp. 2014), s. 11–26. DOI: 10.5121/ijdms.2014.6402.

Příloha A

Obsah přiloženého CD

`zaverecna_prace.pdf` ..... text práce ve formátu PDF
`test_tool.zip` ..... archiv obsahující zdrojové kódy, vstupní data, definici testovacího případu, uživatelskou dokumentaci a návod na vytvoření clusteru za pomoci technologie VirtualBox