VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

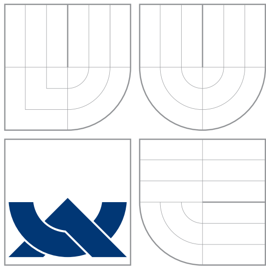
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

IMPLEMENTACE SHLUKOVÁNÍ REGULÁRNÍCH VÝRAZŮ POMOCÍ MAPREDUCE PŘÍSTUPU

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

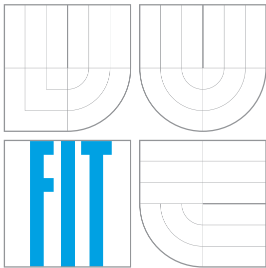
AUTOR PRÁCE — MARTIN ŠAFÁŘ
AUTHOR

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

IMPLEMENTACE SHLUKOVÁNÍ REGULÁRNÍCH VÝRAZŮ POMOCÍ MAPREDUCE PŘÍSTUPU

IMPLEMENTATION OF REGULAR EXPRESSION GROUPING IN MAPREDUCE PARADIGM

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — MARTIN ŠAFÁŘ

AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN KAŠTIL

SUPERVISOR

BRNO 2014

Abstrakt

Hlavním přínosem této práce je návrh a implementace aplikace, která využívá model MapReduce a Apache Hadoop pro urychlení shlukování regulárních výrazů. V této prácí jsou popsány algoritmy, které se využívají pro shlukování regulárních výrazů a je navrženo několik vylepšení pro tyto algoritmy. Experimenty prováděné v rámci této práce ukázaly, že cluster skládající se z 20ti počítačů dokáže oproti klasickému přístupu zrychlit shlukování až desetinásobně.

Abstract

The greatest contribution of this thesis is design and implementation of program, that uses MapReduce paradigm and Apache Hadoop for acceleration of regular expression grouping. This paper also describes algorithms, that are used for regular expression grouping and proposes some improvements for these algorithms. Experiments carried out in this thesis show, that a cluster of 20 computers can speed up the grouping ten times.

Klíčová slova

Intrusion Detection System, MapReduce, Hadoop, Shlukování regulárních výrazů

Keywords

Intrusion Detection System, MapReduce, Hadoop, Regular expression grouping

Citace

Martin Šafář: Implementace shlukování regulárních výrazů pomocí MapReduce přístupu, bakalářská práce, Brno, FIT VUT v Brně, 2014

Implementace shlukování regulárních výrazů pomocí MapReduce přístupu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Kaštila

....
Martin Šafář
21. května 2014

© Martin Šafář, 2014.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 MapReduce **5**
2.1 Co je MapReduce 5
2.2 Princip MapReduce 5
2.3 Průběh výpočtu operací MapReduce 5
2.4 Tolerance chyb 7
2.5 Využití síťového provozu 7

3 Apache Hadoop **8**
3.1 HDFS 8
3.2 Hadoop MapReduce 9
3.3 Plánování úloh v clusteru 10

4 Shlukování regulárních výrazů v rámci vysokorychlostních sítí **12**
4.1 Algoritmus pro shlukování regulárních výrazů 12
4.2 Inkrementální shlukovací algoritmus 13
4.3 Shlukování bez použití algoritmu pro výběr RV do shluku 14

5 Návrh rozdělení shlukovací metody pro funkce Map a Reduce **16**
5.1 Návrh s jedním reducerem 16
5.2 Porovnání shlukovacích algoritmů s jedním reducerem 17
5.3 Návrh aplikace s více reducery 17
5.4 Použití více MapReduce úloh 17
5.5 Možnosti vylepšení návrhu 18
5.6 Vyhodnocení algoritmů 18

6 Implementace **19**
6.1 Hlavní skript 19
6.2 Implementace funckí Map a Reduce 19
6.2.1 Jedna MapReduce úloha 20
6.2.2 Více MapReduce úloh 21
6.2.3 Velké množství krátkých MapReduce úloh 21
6.3 Vylepšení metod shlukování 22
6.3.1 Vylepšení algoritmu popsaného v 4.1 22
6.3.2 Vylepšení inkrementálního algoritmu 22
6.4 Parametry programu a použití 23
6.4.1 Výběr algoritmu pro shlukování 23

6.4.2 Shlukování s použitím Hadoop 23
6.4.3 Stanovení maximálního počtu stavů KA po shlukování 23

7 Testy a hodnocení **24**
7.1 Specifikace testovacích sestav a vytížení jednotlivých komponent 24
7.2 Vytváření grafové struktury a výpočet koeficientu interakce 25
7.3 Porovnání rychlosti a efektivity shlukovacích algoritmů 26
7.4 Výběr regulárních výrazů do shluku s a bez pomoci clusteru 27
7.5 Využití modelu MapReduce pro shlukování regulárních výrazů 28
7.6 Doporučení pro uživatele 28

8 Závěr **30**

Kapitola 1

Úvod

S tím, jak se stále rozšiřuje velikost sítě Internetu a jeho význam v každodenním životě, roste i hrozba kybernetických útoků. Jedním z prostředků, který pomáhá těmto útokům předcházet je analýza uživatelských dat packetu (packet payload) v reálném čase. Analýza se provádí porovnáváním obsahu paketu vůči seznamu regulárních výrazů (RV). Tento způsob je jednou z částí tzv. Systému pro odhalení průniku (Intrusion Detection System - IDS). Princip IDS vyžaduje velice krátký čas odezvy, aby bylo možno útoku efektivně zabránit, nicméně s tím jak se objevují nové způsoby útoků a nové hrozby, musí narůstat i množství pravidel, podle kterých je paket analyzován. Mezi IDS patří například Snort[1], který v současné době obsahuje ve svých pravidlech celkem 6435 unikátních regulárních výrazů [10]. V dubnu 2003 obsahoval 1131 unikátních regulárních výrazů [12].

S narůstajícím počtem pravidel narůstá úměrně i náročnost výpočtů při porovnávání paketů s jednotlivými pravidly, a to jak na procesor, tak i na paměť. Vznikla tudíž potřeba tento proces zefektivnit. Analýza paketu regulárním výrazem spočívá v převedení RV na konečný automat (KA), na kterém se obsah paketu odsimuluje. Pokud automat tento řetězec přijme, znamená to, že paket odpovídá danému pravidlu. Existuje několik možností jak tento proces zrychlit. Jednou z možností je využití programovatelných hradlových polí, které nabízejí vysoký výkon a zároveň možnost kdykoliv pravidla rozšiřovat. Ještě rychlejší by mohly být specializované hardwarové součástky, nicméně nemožnost modifikace procházených pravidel by znamenala výrobu nových čipů při každé úpravě pravidel [6]. Další možností jak zrychlit proces analýzy paketů je snížení počtu pravidel vůči kterým se paket kontroluje. Toho lze dosáhnout pomocí sjednocení RV, respektive sjednocení KA. Sjednocení KA jsme schopni provést vcelku efektivně, nicméně pokud bychom sjednotili všechny do jednoho, paměťové nároky by byly neúnosné (pro velký počtu stavů a přechodů). Proto je nutné správně vybrat, které regulární výrazy, respektive které konečné automaty je vhodné spojit a které ne. Tímto problémem se zabývají články [12] a [1].

Cílem této práce je prozkoumat využití modelu MapReduce a distribuovaných výpočtů pro urychlení shlukování regulárních výrazů, a to při zachování efektivity navržených algoritmů, tak, aby byl celý proces škálovatelný. Dalším úkolem je vytvořit návrh aplikace v Hadoopu a tento návrh implementovat.

Kapitola 2 by měla čtenáře seznámit se základním principem fungování modelu MapReduce a s principem fungování počítačového clusteru. Navíc zmiňuje některé problémy, které musí implementace tohoto modelu řešit.

Kapitola 3 popisuje několik modulů patřicích pod Apache Hadoop (implementaci mo-

[1] <http://www.snort.org/>

delu MapReduce). Především je zmíněn HDFS (Hadoop Distributed Filesystem), MapReduce a YARN. Tyto moduly jsou páteří distribuovaných výpočtů pod Apache Hadoop.

V kapitole 4 jsou popsány a porovnány tři algoritmy pro shlukování regulárních výrazů, pseudokódem je naznačeno jejich fungování a jsou zmíněny jejich výhody a nevýhody.

V kapitole 5 jsou uvedeny návrhy jak bude vypadat výstupní aplikace této práce a jsou porovnány tři různé přístupy k využití modelu MapReduce a frameworku Hadoop.

V předposlední kapitole je popsána implementace výsledné aplikace a poslední kapitola hodnotí výsledky získané z testování a porovnává jednotlivé přístupy popsané v kapitole 7.

Kapitola 2

MapReduce

Tato kapitola popisuje programovací model MapReduce, jeho vznik, princip fungování a využití. Článek o MapReduce byl publikován pod společností Google jejími zaměstnanci J. Deanem a S. Ghemawatem v roce 2004 [3].

2.1 Co je MapReduce

MapReduce je programovací model a s ním spojená implementace pro zpracovávání a generování velkých souborů dat. Výpočty probíhají zpravidla na tzv. clusteru (vetší množství komoditních strojů, které jsou propojeny v síti). Motivací k vytvoření MapReduce byla opakovaná nutnost provádět různé operace s velkým množstvím dat v rozumném čase (např. procházení dokumentů, či indexování webových stránek pomocí dat z crawlerů, atd.). Tyto operace byly většinou prováděny paralelně na stovkách počítačů a bylo nutné věnovat hodně úsilí nejen samotnému problému, ale zároveň i implementaci správy dílčích výpočtů a shromažďování výsledků. MapReduce je abstrakce problému, která pomáhá uživateli soustředit se na řešení samotného problému bez nutnosti explicitní implementace tolerance chyb, paralelizace nebo distribuce dat [3].

2.2 Princip MapReduce

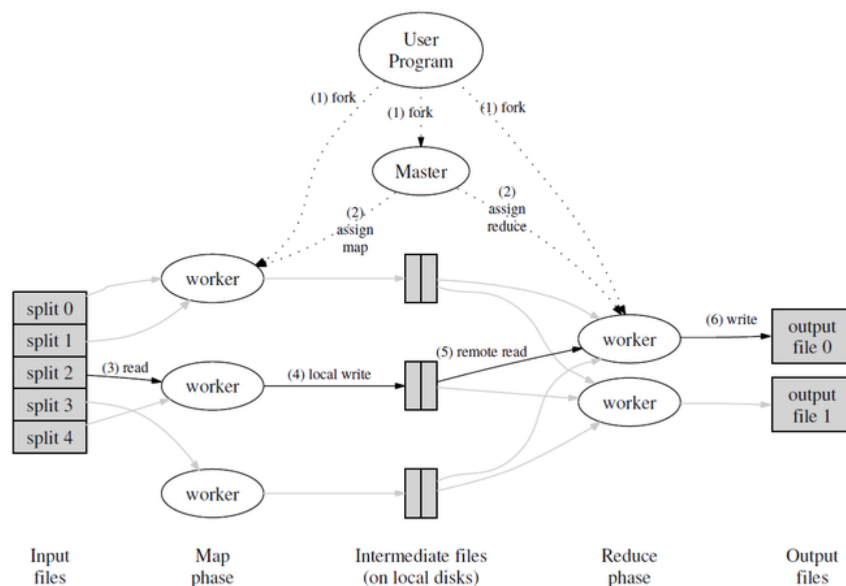
Základním principem fungování tohoto modelu je rozdělení hlavní práce s daty na dvě uživatelem definované funkce. Jsou to funkce Map a Reduce (tyto funkce také dávají jméno celému modelu). Funkce Map bere na vstupu dvojice klíč-hodnota a jejím výstupem je seznam průběžných dvojic klíč-hodnota. Tyto hodnoty jsou podle klíčů seskupeny, seřazeny a předány funkci Reduce. Ta přijímá seskupené hodnoty podle klíčů, přičemž pro každý klíč proběhne právě jednou a vrátí výsledek uživateli.[3]

$$map(k1, v1) \rightarrow list(k2, v2)$$

$$reduce(k2, list(v2)) \rightarrow list(v2)$$

2.3 Průběh výpočtu operací MapReduce

Při zavolání úlohy MapReduce proběhne postupně několik kroků (na obrázku 2.1 jsou kroky očíslovány) [3]:



Obrázek 2.1: Schéma průběhu provádění MapReduce úlohy (převzato z [12])

1. Nejprve daná implementace MapReduce rozdělí vstup na M částí, podle druhu úlohy se velikost těchto částí může pohybovat od stovek Bytů až po desítky MB. Rozdělená data a uživatelská část zdrojových kódů jsou zpřístupněny ostatním uzlům, například přes distribuovaný souborový systém.

2. Jeden z uzlů je určen jako hlavní (master) a ten přiděluje úkoly všem ostatním (worker - pracovník). Celkem bude provedeno M volání funkce Map a R volání funkce Reduce, přičemž poměr M ku R může být až řádově 1000 : 1, nicméně zpravidla je vždy $M > R$. Pokud některý z workerů dokončí svůj úkol, přidělí mu master další.

3. Tzv. mapper postupně zpracovává přidělený vstup a provádí nad ním funkci Map, která postupně ukládá výsledek do paměti.

4. Výstupy z mapperů jsou pravidelně ukládány na lokální disk uzlu. Po zpracování celého vstupu jsou data roztříděna do různých front pro reducery a informace o jejich umístění jsou odeslány masterovi. Tyto informace jsou poté předávány uzlům provádějícím funkci Reduce (tzv. reducery).

5. Když reducer dostane informace o umístění výstupů mapperů, postupně od všech posbírá data.

6. Ve chvíli, kdy má reducer všechna potřebná data, seřadí je podle klíčů a zavolá uživatelem definovanou funkci Reduce(). Funkce Reduce() zpravidla nemůže být spuštěna, dokud neproběhly všechny úlohy mapperů.

7. Po provedení všech map a reduce úloh vrátí master řízení programu, ze kterého byl volán a předá mu umístění výsledků z jednotlivých reducerů.

Během celého procesu uchovává hlavní uzel několik informací o jednotlivých operacích. Pro každou úlohu map a reduce zaznamenává její stav (nečinná, probíhající a hotová) a současně také uzel, který na úloze pracuje (případně pracoval) [3].

2.4 Tolerance chyb

Protože MapReduce může fungovat i na několika stovkách strojů, je nutné počítat s tím, že může docházet k chybám, ať už v samotném provádění úloh, tak i v rámci komunikace po síti.

Všichni pracovníci jsou pravidelně kontaktováni masterem, za účelem kontroly dostupnosti. Pokud pracovník po nějakou dobu neodpovídá, označí si ho master jako chybový. Úloha, kterou uzel právě provádí, je označena jako nezpracovaná a je znovu zadána některému ze zbývajících pracovníků [3].

Pro ochranu proti chybě hlavního uzlu je možné pravidelně ukládat jeho průběžný stav a při selhání procesu pouze obnovit běh na jiném stroji ze zálohovaných dat. Protože je hlavní uzel pouze jeden, je šance, že dojde k chybě velmi malá a tudíž v některých implementacích není tato chyba ošetřena. Při chybě hlavního uzlu jednoduše celý proces skončí a uživatel musí znovu zadat úlohu [3].

Navíc může docházet k náhodným chybám, například z důvodu poškození disku, CPU, nebo operační paměti. K odhalení těchto chyb ale není MapReduce uzpůsoben. Přestože většina výpočtů pro které je tento model využíván je schopna tolerovat takovéto chyby (na celkovém výsledku se projeví jen minimálně), vyskytují případy, kdy je třeba dosáhnout co nejvyšší přesnosti při výpočtech. O způsobech, jak těmto chybám předejít se nachází více v článku [2].

2.5 Využití síťového provozu

V poměru k výpočetní síle clusteru bývá jeho velkou slabinou síťový přenos. Cluster může fungovat i přes několik datových center a přenášet celý vstup mezi jednotlivými centry by zásadně vytížilo síť. Z tohoto důvodu je vhodné použít některý z distribuovaných souborových systému. Implementace Hadoop například využívá HDFS, dalšími příklady jsou GFS - Google File System, nebo Ceph. Tyto souborové systémy implicitně ukládají soubory na více místech najednou (typicky 3 různé uzly). Hlavní uzel se pokouší přidělit mapovací úlohu pracovníkům primárně tak, aby pracovali s daty, která se nacházejí na jejich vlastních discích. V případě, že toto není možné, přidělí master úlohu pracovníkovi, který je u dat „nejblíže“. Vzdálenost mezi dvěma uzly je určována podle kapacity a vytížení síťového spojení mezi nimi. Tyto informace si musí uchovávat master [3].

Kapitola 3

Apache Hadoop

Apache Hadoop je framework implementovaný v jazyce Java, skládající se z několika různých modulů a projektů. Nejznámější z nich jsou MapReduce, implementace stejnojmeného modelu a Hadoop Distributed File System (HDFS) - distribuovaný souborový systém. Autorem frameworku je Doug Cutting, který se inspiroval článkem od J. Deana a S. Ghemawata a jejich implementací MapReduce (patřící pod Google), a stejně tak i distribuovaným souborovým systémem Google File System (GFS). Hadoop začal vznikat jako podprojekt open-sourcového internetového vyhledávače Nutch a nakonec byl v roce 2008 povýšen na jeden z hlavních projektů Apache a projekt NDFS (Nutch Distributed Filesystem) byl přejmenován na HDFS. Celý projekt Apache Hadoop je open-sourcový pod licencí „Apache License, Version 2.0 “ [1] [8]. Většina hlavních podprojektů spadá primárně pod Apache Hadoop Foundation, nicméně jak se celá rodina modulů postupně rozrůstá, vzniká spousta samostaných projektů doplňujích funkčnost, případně zvyšujících efektivitu či zajištění vyšší tolerance pro náhodné chyby. Kromě dvou zmíněných modulů patří pod Apache Hadoop také modul Common - Sada nástrojů a rozhraní pro distribuované souborové systémy a obecné vstup/výstupní operace (serializace dat, Java Remote Procedure Call,...) a YARN (Yet Another Resource Negotiator - „Ještě jeden správce zdrojů“), což je distribuovaný operační systém pro správu clusterů [8].

3.1 HDFS

HDFS je distribuovaný souborový systém, navržený a implementovaný tak, aby mohl fungovat na clusteru komoditních počítačů. Díky implementaci v jazyce Java je multiplatformní a oproti ostatním distribuovaným souborovým systémům se vyznačuje především vysokou tolerancí k chybám na jednotlivých uzlech clusteru. Aplikace využívající HDFS pracují většinou se soubory o velikosti v řádech gigabytů až terabytů a s celkovou kapacitou clusteru dosahující až desítek petabytů (v době psaní této práce měl jediný HDFS cluster patřící Facebooku celkovou kapacitu v řádu stovek petabytů [11]). Proto je tento souborový systém navržený spíše pro co největší propustnost dat, než pro nízkou odezvu při požadavcích aplikace [8]. HDFS se skládá z jednoho hlavního serveru - NameNode a datových serverů - DataNode. NameNode má na starosti správu souborového systému a obstarává přístup k souborům při požadavcích od uživatele. Protože NameNode je jediné místo, kde jsou uložena metadata k souboru a umístění jednotlivých částí souboru, je nutné mít sekundární NameNode, který neustále odebírá od hlavního serveru obraz souborového

[1] <http://www.apache.org/licenses/LICENSE-2.0.html>

systému a v případě chyby může pomoct při jeho restartování, nicméně jeho funkci nemůže plně nahradit. Další možností, kterou poskytuje HDFS, je neustálé ukládání perzistentních dat souborového systému na nezávislé úložište, odkud je možné v případě chyby obnovit poslední stav před chybou [5]. DataNode zaštiťuje a spravuje souborový systém na jednotlivých uzlech clusteru a bývá většinou jeden na každý uzel clusteru. Pravidelně posílá informace hlavnímu uzlu o stavu svého úložiště a o blocích, které ukládá. Soubory ukládané do HDFS jsou rozdělovány podle nastavení na bloky o velikosti v řádech desítek megabytů. Každý z těchto bloků je většinou uložen na více uzlech, ve výchozím nastavení na třech uzlech. O tom, na které uzly se jednotlivé bloky uloží, rozhoduje NameNode a většinou se ohlíží jak na bezpečnost uložení (aby nebyly všechny bloky na jednom uzlu), tak i na vytížení sítě (snaží se vyhnout odesílání několika kopií bloku např. mezi datovými centry). Ideálním řešením například pro 3 kopie bloku je 2 kopie na jednom uzlu, ale na různých fyzických úložištích a jedna na jiném uzlu, který se nachází „blízko“ k tomu prvnímu (Vzdálenost uzlů od sebe si monitoruje NameNode a určuje ji podle propustnosti sítě mezi dvěma uzly) [8] [5].

3.2 Hadoop MapReduce

Modul MapReduce může díky Hadoop Streaming API provádět programy napsané nejen v jazyce Java, ale zároveň i v různých dalších jazycích (např. Python, Ruby, C++ a další). Tento modul se podobně jako HDFS skládá z jednoho hlavního prvku - JobTracker a většího počtu TaskTrackerů (například jeden TaskTracker na každé jádro procesoru jednotlivých strojů). JobTracker přijímá výpočetní úkoly od uživatelské aplikace a rozděluje je volným TaskTrackerům v clusteru. Pokud je na uzlu, kde jsou zároveň data, volný TaskTracker, je mu přidělen daný výpočetní úkon. V opačném případě se hledá nejbližší TaskTracker v rámci datového centra a v nejhorším případě musí být data odeslána do jiného datového centra a úkon proveden tam. Obecně se celý framework Hadoop snaží maximálně omezit využití sítě a to z toho důvodu, že mezi množstvím dat, která jsme schopni uložit a rychlostí jakou můžeme data posílat po síti, je propastný rozdíl [8]. Pokud dojde k chybě při provádění úlohy, přidělí JobTracker úlohu nějakému jinému uzlu. Zároveň také pravidelně jednou za několik minut posíla TaskTracker JobTrackeru informace o svém stavu.

Na obrázku 3.1 je znázorněný průběh plnění MapReduce úlohy zadané uživatelem. V celém procesu figurují celkem čtyři hlavní entity. Klient, který žádá o provedení určitého úkolu. Dále JobTracker a TaskTrackery, které jak je zmíněno výše zajišťují běh aplikace a samotné výpočty. A jako poslední distribuovaný souborový systém. Provádění dané úlohy se skládá z následujících kroků [8]:

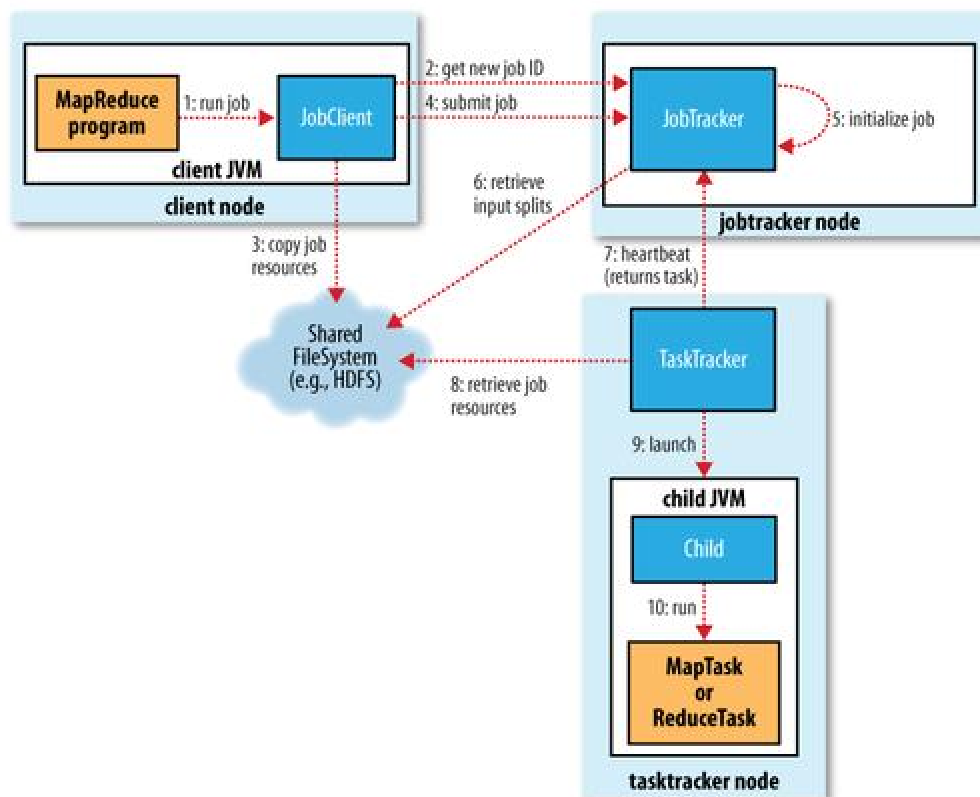
1. Klient zadá z uživatelské části programu příkaz k provedení MapReduce úlohy voláním metody runJob() ve třídě JobClient.

2. JobClient nechá třídu JobTracker vygenerovat nové ID pro danou úlohu.

3. Dojde k rozkopírování uživatelské části programu do souborového systému včetně zdrojových kódů a vstupu programu, již rozděleného na jednotlivé části. Zdrojové kódy jsou kopírovány s vysokou mírou duplicity, protože k nim pravděpodobně nakonec budou přistupovat všechny uzly v clusteru.

4. JobClient dá JobTrackeru vědět, že je vše připraveno a mohou začít výpočty. Po zadání práce se JobClient každou vteřinu dotazuje na stav výpočtů.

5. Dojde k inicializaci a vytvoření obalové třídy, která reprezentuje provádění dané úlohy a drží informace o průběhu výpočtu.
6. Aby mohl JobTracker vytvořit frontu s jednotlivými podúlohami a začít rozdělovat práci, musí nejdřív zjistit informace o počtu a velikosti jednotlivých částí, na které byl rozdělen vstup.
7. Všechny TaskTrackery pravidelně kontaktují JobTracker a dávají mu vědět, zda provádí výpočty, nebo zda jsou volní. Pomocí návratové hodnoty jim JobTracker může přidělit úlohu k provedení.
8. Po přijetí úkolu si TaskTracker dohledá příslušný vstup a zdrojové kódy aplikace.
9. Dojde k vytvoření nového potomka JVM.
10. Potomek začne postupně provádět všechny zadané úlohy.



Obrázek 3.1: Schéma průběhu provádění MapReduce úlohy ve frameworku Hadoop(převzato z [8])

3.3 Plánování úloh v clusteru

V původní verzi Hadoop existoval pouze jednoduchý plánovač, který prováděl zadané úkoly od uživatele pomocí jednoduché fronty. Tento způsob není ideální hlavně proto, že není využit výpočetní potenciál celého clusteru. Na druhou stranu při sdílení výpočetních zdrojů

mezi několika uživateli je třeba zajistit, aby se každý uživatel dostal k výpočetnímu času a nedocházelo k vyhladovění některých úloh. Hadoop dovoluje nastavit jednotlivým úlohám priority podle kterých se plánovač řídí, nicméně bez preempce stále může jedna dlouhá úloha s nízkou prioritou zablokovat krátkou úlohu s vysokou prioritou. Ve výchozím nastavení používá Hadoop právě frontový plánovač. Nabízí ale zároveň další plánovače: Fair Scheduler a Capacity scheduler [8]. Fair scheduler, vyvíjený ve společnosti Facebook [2], má jednoduchou filozofii. Jeho základní myšlenkou je přidělit každé úloze stejný procesorový čas a tudíž vyhovět jak úlohám, kterým stačí jen krátká doba na dokončení, tak i dlouho trvajícím úlohám, které by mohly kvůli nízké prioritě hladovět. Capacity scheduler, vyvíjený firmou Yahoo! Inc. [3], dělí aktivní úlohy do několika prioritních front, kde každá fronta dostává přidělený výpočetní čas a ten si mezi sebe úlohy rozdělí podle priority. Plánovač může jednotlivým frontám přidávat nebo ubírat výpočetní kapacitu podle toho, zda ji jsou schopny využívat, nebo ne.

[2] `<http://www.facebook.com/>`
[3] `<http://www.yahoo.com/>`

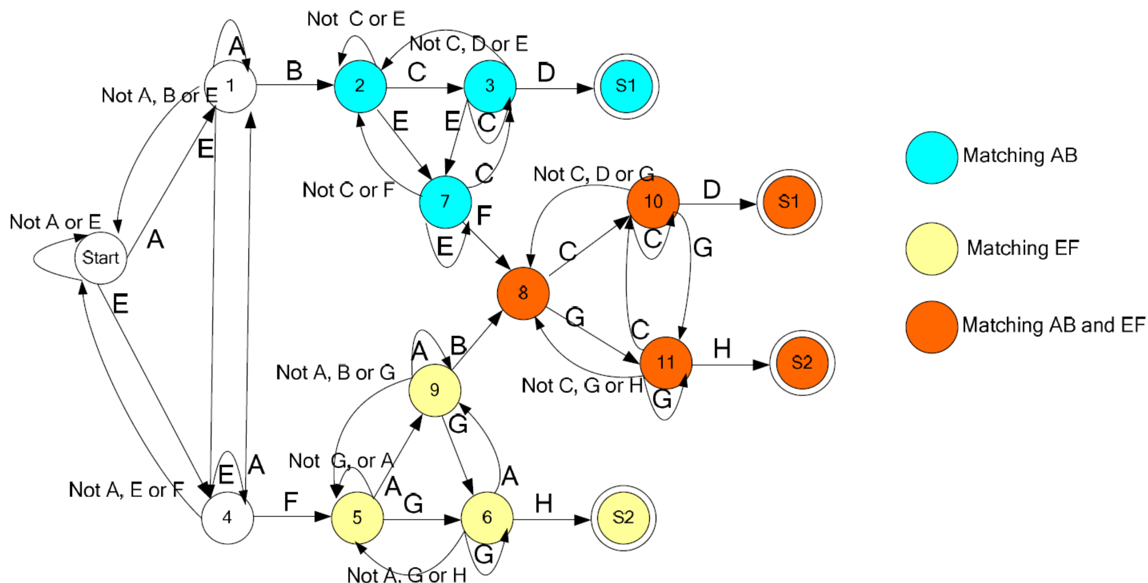
Kapitola 4

Shlukování regulárních výrazů v rámci vysokorychlostních sítí

Paměťové nároky na analýzu toku dat v síti pomocí porovnávání obsahu paketu vůči seznamu pravidel reprezentovaných regulárními výrazy jsou velice vysoké. Navíc může tento proces kriticky zatěžovat procesor a výrazně tím snížit propustnost sítě. Jednou z možností jak snížit náročnost výpočtu je shlukování regulárních výrazů. Proces shlukování se skládá z převodu regulárních výrazů na nedeterministické konečné automaty (NKA), jejich následné sjednocení, determinizace a minimalizace. Je zřejmé, že spojením n regulárních výrazů do jednoho dojde ke snížení složitosti algoritmu z O(n) na O(1) pro zpracování jednoho znaku. Bohužel při shlukování některých regulárních výrazů může dojít až k exponenciálnímu nárůstu na velikosti výsledného automatu [12]. Například automat. který vzniká sjednocením regulárních výrazů „.*AB.*CD“ a „.*EF.*GH“. Jak je vidět na obrázku 4.1, po sjednocení automatů došlo k navýšení počtu stavů téměř o polovinu oproti součtu stavů z automatů před sjednocením. Je totiž nutné počítat s oběma prefixy, jak „AB“, tak „EF“, a jen kvůli ošetření těchto případů přibyly stavy 8, 10 a 11. Obecně je tento problém způsoben neurčitou délkou prefixu a narůstá exponenciálně velikost automatu. S každým dalším automatem se velikost výsledného automatu zvyšuje tolikrát, kolikrát se v regulárních výrazech opakuje výraz „.*“. Stejnou rychlostí se zvětšují také paměťové nároky. (Viz. [12]).

4.1 Algoritmus pro shlukování regulárních výrazů

Aby bylo shlukování efektivní, je třeba vybírat regulární výrazy tak, aby nevznikaly situace popsané výše. V [12] je popsán algoritmus, který z n regulárních výrazů udělá k shluků, čímž složitost zpracování jednoho písmene se sníží z O(n) na O(k) bez zvýšení paměťové náročnosti. Pro popis algoritmu si nejdříve definujeme interakci. Dva výrazy jsou v interakci, pokud počet stavů automatu po jejich sjednocení je větší než součet stavů původních automatů. Počet stavů počítáme vždy po převedení na minimální deterministický automat. Pomocí této relace mezi výrazy budeme následně vytvářet skupiny regulárních výrazů a ty poté shlukovat. Nejprve stanovíme, limit do kterého se musí vejít velikost výsledného konečného automatu. Poté vytvoříme graf, kde pro každý regulární výraz budeme mít jeden uzel. Hrana mezi dvěma uzly bude tehdy, pokud jsou dva odpovídající regulární výrazy v relaci. Když máme graf zkonstruovaný, vybereme uzel, do kterého směřuje nejméně hran a vložíme ho do nové skupiny. Dále postupně vždy vybíráme takový uzel, ze kterého směřuje co nejméně hran do uzlů, které již v nové skupině jsou. Takový uzel přidáme do nové



Obrázek 4.1: „Exploze stavů“ při sjednocení dvou automatů (převzato z [12])

skupiny. Pokud počet stavů v automatu, který vznikne sjednocením regulárních výrazů ze zmíněné skupiny, přesáhne stanovenou hraniční hodnotu, odstraníme z grafu všechny uzly, které jsou v nové skupině a tuto skupinu vyprázdníme. Tento proces opakujeme dokud jsou v grafu nejaké uzly.

4.2 Inkrementální shlukovací algoritmus

Inkrementální shlukovací algoritmus je podobný algoritmu 4.1. Liší se především v konstrukci grafu. Místo obyčejného neorientovaného grafu používá druhý algoritmus ohodnocený souvislý graf. Ohodnocení hrany vyjadřuje míru interakce mezi dvěma regulárními výrazy a vypočítává se podle následujícího vzorce:

$$Mira\ interakce = \frac{(pocet\ stavu\ KA\ po\ sjednoceni\ uzlu)}{(soucet\ poctu\ stavu\ puvodnich\ KA)}$$

Jak napovídá tento vzorec, pokud je výsledné ohodnocení rovno 1, nedošlo k žádné změně počtu stavů. V případě, že ohodnocení hrany je menší než 1, respektive větší než 1, znamená to, že se počet stavů oproti původnímu počtu po sjednocení KA snížil, respektive zvýšil [1].

Inkrementální algoritmus na rozdíl od algoritmu 1 využívá to, že informace o míře interakce mezi dvěma uzly není binární, jak je tomu u 1, ale je určena desetinným číslem. Inkrementální algoritmus proto nemusí vybírat náhodně ze skupiny RV, do jejichž uzlů vede nejméně hran, ale může si všechny výrazy seřadit podle míry interakce. U tohoto algortimu dochází také pravidelně k přepočítávání ohodnocení všech uzlů vůči nově vytvářené skupině. To je výhodnější z hlediska efektivity shlukování (viz [1]), protože vždy hledáme ten nejvhodnější výraz ke sjednocení, nicméně počet determinizací se zvýší na dvojnásobek - pro N RV je nutno operaci sjednocení provést $\frac{(N^2)-N}{2}$ pro vytvoření základního grafu a poté ještě $\frac{(N^2)}{2}$ při přepočítávání ohodnocení. Článek [7] navrhuje další způsob ohodnoco-

Algoritmus 1 Algoritmus pro shlukování regulárních výrazů (převzat z [12])

for regulární výraz R_i ze seznamu **do**
    for regulární výraz R_j ze seznamu **do**
        vypočítej koeficient interakce $KI = interaction(R_i, R_j)$
    end for
end for
Vytvoř graf $G(V, E)$, kde:

- V je množina uzlů a každý uzel V_i odpovídá regulárnímu výrazů R_i.
- E je množina hran grafu a vrchol $(V_i, V_j) \in E \Leftrightarrow interaction(R_i, R_j) > 1$

while $V \neq \emptyset$ **do**
    Vytvoř prázdou množinu NG
    Vyber uzel, ze kterého vede nejmenší počet hran a přidej jemu odpovídající RV R_i do množiny NG a převeď R_i na minimální DKA A_m
    while $V \neq NG$ **do**
        Vyber uzel V_x, ze kterého vede nejméně hran do uzlů v NG
        Vytvoř sjednocení automatů $A_m \cup R_x$ a převeď na minimální DKA.
        if počet stavů $poetStav(A_m \cup R_x) > limitpotustav$ **then**
            break;
        else
            přidej R_x do NG
        end if
    end while
    Odeber všechny uzly v množině NG z množiny V
end while

vání. V zásadě se dost podobá způsobu z [1], ale ohodnocení hrany vypočítává podle jiného vzorce. Výsledná hodnota určuje míru podobnosti dvou regulárních výrazů a může být buď záporná, nebo mezi 0 a 1. Záporná hodnota znamená, že si automaty nejsou podobné, hodnota mezi 0 a 1, že si podobné jsou (obecně čím vyšší číslo, tím větší podobnost) a hodnocení hrany rovno 1 znamená, že sjednocení odpovídajících RV nezvýší počet stavů výsledného automatu.

4.3 Shlukování bez použití algoritmu pro výběr RV do shluku

Další možností jak shlukovat regulární výrazy je využít informace získané při vytváření grafové struktury (koeficienty interakce pro všechny dvojice RV). Je možné místo výběru regulárního výrazu, který má být přidán do shluku, seřadit regulární výrazy předem a shlukovat je popořadě. Experimentálně jsem při implementaci výsledného nástroje zjistil, že při seřazení seznamu regulárních výrazů podle jejich průměrného koeficientu interakce je možné dosáhnout podobné efektivity jako u algoritmů popsaných výše (více v sekci 7.3).

Tento přístup by bylo možné zdokonalit pomocí analýzy samotných regulárních výrazů a ne pouze podle automatů z nich vytvořených . Díky takovému zpracování by bylo možné rozdělit předem regulární výrazy do několika skupin. Taková situace by současně umožňovala využít model MapReduce pro shlukování a tím ho zrychlit.

Algoritmus 2 Inkrementální algoritmus pro shlukování regulárních výrazů (převzat z [1])

for regulární výraz R_i ze seznamu **do**
  for regulární výraz R_j ze seznamu **do**
    vypočítej koeficient interakce $KI = interaction(R_i, R_j)$
  end for
end for
Vytvoř graf $G(V, E)$, kde:

- V je množina uzlů a každý uzel V_i odpovídá regulárnímu výrazů R_i.
- E je množina hran grafu a jejich ohodnocení se vypočítá takto: $w(V_i, V_j) = KI$

while $V \neq \emptyset$ **do**
  Vytvoř prázdou množinu NG
  Vyber uzel, který má nejnižší průměrný koeficient interakce, přidej jemu odpovídající regulární výraz R_i do množiny NG, převeď R_i na minimální DKA A_m a přejmenuj uzel V_i na V_{A_m}
  while $V \neq \emptyset$ **do**
    Vyber uzel V_x, který má nejnižší interakci vůči V_{A_m}
    Vytvoř sjednocení automatů $A_m \cup R_x$ a převeď na minimální DKA.
    if $pocetStavu(A_m \cup R_x) > limitpoctustavu$ **then**
      break;
    else
      přidej R_x do NG
      $A_m = A_m \cup R_x$
      Odstraň V_x a z G
      aktualizuj ohodnocení hran v grafu G
    end if
  end while
  Odeber všechny uzly v množině NG z množiny V
end while

Kapitola 5

Návrh rozdělení shlukovací metody pro funkce Map a Reduce

Cílem bakalářské práce je navrhnout a implementovat shlukování regulárních výrazů pomocí modelu MapReduce. V této kapitole jsou porovnány algoritmy zmíněné v kapitole 4, jsou zmíněny jejich teoretické výhody a nevýhody při využití frameworku Hadoop a je navržena výsledná aplikace nejprve s jedním reducerem, následně s více reducery a nakonec s využitím více úloh MapReduce. Celá aplikace bude implementována v jazyce Python s využitím Hadoop Streaming API. Pro převádění regulárních výrazů na konečný automat a následné sjednocení, determinizaci a minimalizaci automatu bude využit nástroj NetBench[1] a celá aplikace bude obalena skriptem v jazyce Python.

5.1 Návrh s jedním reducerem

V obou algoritmech z kapitoly 4 je potřeba nejprve provést porovnání regulárních výrazů každý s každým, což je pro N regulárních výrazů celkem $\frac{(N^2-N)}{2}$ sjednocení automatů. Proto bude právě tento proces probíhat v mapovací funkci. Na jeho vstupu bude seznam dvojic regulárních výrazů - každý s každým ze vstupního seznamu shlukovaných regulárních výrazů. Nad těmito dvojicemi provede mapper sjednocení automatů a spočítá poměr počtů uzlů sjednoceného automatu a součet počtů před sjednocením. Na jeho výstupu v případě použití algoritmu 1 bude jeden z regulárních výrazů klíč (na tom, který bude v klíči a který v hodnotě nezáleží, protože je jen jeden reducer a ten musí posbírat všechny výsledky najednou do výsledného grafu) a druhý hodnota v případě, že jsou tyto uzly v relaci. Pro algoritmus 2 bude navíc ve výstupu vypočítaná míra interakce podle vzorce v 4.2.

Všechny shromážděné výsledky poté bude reducer procházet a ukládat je do grafové struktury. Z grafu budou postupně odebírány uzly a přidávány do nové skupiny, podle jednoho z algoritmů v 4. Na výstupu reduceru už konečné automaty reprezentující shluky regulárních výrazů.

Výhodou návrhu s jedním reducerem je jednoduchost a především efektivita procesu výběru regulárních výrazů do nově vytvářenéhé shluku (oproti návrhu s více reducery, respektive více MapReduce úlohami). Velkou nevýhodou je nutnost držet v paměti celou strukturu grafu při shlukování, což nemusí být problém při shlukování menších počtů regulárních výrazů (řádově stovky), ale například pro 6000 regulárních výrazů mohou paměťové

[1] `<http://merlin.fit.vutbr.cz/ant/netbench/index.html>`

nároky výrazně narůst. Navíc vhledem ke škálovatelnosti provádění výpočtů v mapperech, která je téměř neomezená, by mohl reducer značně zpomalit provádění celého výpočtu.

5.2 Porovnání shlukovacích algoritmů s jedním reducerem

V článku [1] bylo testováním zjištěno, že algoritmus 2 dosahuje při shlukování RV přibližně o 10% lepších výsledků v počtu stavů a shluků sjednocených konečných automatů oproti algoritmu 1. Je ale výpočetně náročnější a pro návrh s jedním reducerem se nehodí z důvodů popsaných v kapitole 4. Pro jeden reducer se v zásadě hodí použít jen algoritmus 1, případně přístup popsaný v 4.3.

5.3 Návrh aplikace s více reducery

Bez předchozího zpracování nejsme schopni výstup mapperů efektivně rozdělit mezi více reducerů, protože potřebujeme ucelenou informaci o interakci mezi regulárními výrazy. Článek [12] nabízí některé možnosti jak předem rozpoznat, které regulární výrazy by mohly při shlukování způsobit exponenciální nárůst počtu stavů, nicméně pro naši potřebu jsou tyto návrhy nedostačující. Bylo by možné se pokusit předem analyzovat všechny regulární výrazy bez vzájemného porovnávání a rozdělit je do skupin podle podobnosti, v takovém případě už by bylo ale efektivnější použít více MapReduce úloh, protože celý výstup jedné map úlohy by vždy mířil do stejného reduceru. Tato práce se ale vývojem nástroje, který by byl schopen rozdělit regulární výrazy podle podobnosti zabývat nebude a bude spíše soustředěna na využití distribuovaných výpočtů pro shlukování.

5.4 Použití více MapReduce úloh

Jak je zmíněno výše, využití reducerů pro závěrečnou fázi algortimu nám znemožňuje rozumnou škálovatelnost a tím by použití modelu MapReduce ztrácelo smysl. Pro efektivní využití clusteru bude celý výpočet rozdělen do dvou funkcí Map(). V první úloze mapper podobně jako v předchozích návrzích zjistí míru interakce pro všechny dvojice RV ze vstupního seznamu a reducer posbírá výstupy mapperů a ke každému regulárnímu výrazu vypíše navíc jeho průměrnou míru interakce. Tento seznam přijde po seřazení vzestupně podle míry interakce na vstup druhé MapReduce úlohy. Hadoop nabízí možnost nerozdělovat vstup podle množství dat, ale podle souborů. Tento způsob by se dal použít k rozdělení velkého množství RV na menší skupiny, které by poté v mapperu byly shlukovány podle jednoho ze zmíněných algoritmů, nicméně ve výsledku by zpravidla došlo k úbytku na efektivitě samotného shlukování, protože by často nebyl využit maximální limit pro počet stavů u konečných automatů. Výstupem mapperu už by byly shlukované regularní výrazy, které by už druhý reducer pouze předal na výstup.

Možností jak eliminovat „zbytky“ regulárních výrazů, které nedosáhnou stanoveného limitu, je posílat tyto výrazy do reduceru s příznakem, že stále nejsou sjednoceny a reducer může tyto výrazy ukládat a nakonec nad nimi provést celý postup znovu a to už s využitím MapReduce úlohy, nebo bez ní.

5.5 Možnosti vylepšení návrhu

Pro optimalizaci poslední zmíněné metody lze rozdělit výstup první MapReduce úlohy pomocí heuristiky místo rozdělení podle řádků. Například podle průměrného koeficientu interakce vůči regulárním výrazům ve skupině a podle počtu stavů v determinizovaném automatu můžeme odhadnout kolik regulárních výrazů zařadit do které skupiny. V rámci této práce se pokusím položit alespoň základy pro takový algoritmus.

5.6 Vyhodnocení algoritmů

Jak je zmíněno výše, návrh z 5.4 bude teoreticky o něco méně efektivní ve shlukování oproti původnímu provedení bez MapReduce, reálné výsledky budou popsány v kapitole 7. Na druhou stranu návrh se dvěma MapReduce úlohami je ve všech částech škálovatelný, takže i pro větší množství regulárních výrazů by neměl mít problém s výpočty.

Kapitola 6

Implementace

K implementaci nástroje pro shlukování regulárních výrazů byl použit framework Apache Hadoop[1] a pomocí Java Streaming API jsou volány funkce Map a Reduce, implementované v jazyce Python 2.7. Pro převod regulárních výrazů (zapsaných v syntaxi PCRE [4]) na nedeterministický konečný automat (NKA) a k následné determinizaci a minimalizaci byl využit nástroj NetBench [9]. Determinizace a minimalizace jsou implementovány v jazyce Python. V případě převodu RV na NKA zajišťuje Python pouze rozhraní a převod provádí nástroj implementovaný v jazyce C. Výstupní program této práce je potom obalený skriptem v jazyce Python.

6.1 Hlavní skript

Hlavní skript zpracovává uživatelské parametry programu a zajišťuje zpracování vstupu před voláním úlohy MapReduce. Zároveň informuje uživatele o stavu aplikace. Vstupem tohoto skriptu je textový soubor se seznamem RV v syntaxi PCRE oddělených znakem nového řádku. Tento vstup je pomocí modulu $c$omb.py rozgenerován na seznam dvojic regulárních výrazů oddělených znakem nového řádku a jednotlivé dvojice jsou od sebe odděleny tabulátorem. Pro N regulárních výrazů na vstupu vznikne seznam $\frac{N^2-N}{2}$ dvojic. Tento seznam poté skript přesune do HDFS a spustí podle zadaných parametrů dané úlohy MapReduce.

PCRE syntaxe dovoluje i RV, které popisují neregulární jazyky. Takové RV nelze převést na KA a proto je třeba je ze vstupu odfiltrovat. K tomu slouží modul $p$creCheck.py. Ten zkusí pomocí nástroje netbench převést všechny RV na KA a vrátí pouze ty, které se povede převést. Při testování jsem narazil na některé části výrazů, které způsobovaly zacyklení nástroje netbench. Jednalo se především o třídy popsané rozpětím znaků [\x80-\xff] - např /(USER—PASS)[\x80-\xff]/. V ideálním případě by si tedy měl uživatel sám zkontrolovat, zda všechna pravidla v seznamu popisují regulární jazyk a lze je převést pomocí nástroje NetBench na KA.

6.2 Implementace funckí Map a Reduce

Jak je již zmíněno v kapitole s návrhem, vytvořil jsem dvě verze nástroje. První verze spouští jednu MapReduce úlohu a je škálovatelná pouze v části výpočtu koeficientu interakce dvojic regulárních výrazů. Druhá verze spouští minimálně 2 MapReduce úlohy a je

[1] <http://hadoop.apache.org/>

plně škálovatelná. Uživatel si pomocí parametrů aplikace může vybrat který z návrhů chce použít.

6.2.1 Jedna MapReduce úloha

V tomto návrhu Mapper načítá po řádcích dvojice regulárních výrazů a každý z nich převede pomocí nástroje pcre_parser() z frameworku Netbench na nedeterministický konečný automat. Oba takto vzniklé automaty jsou determinizovány a minimalizovány (opět pomocí frameworku Netbenech). Následně je vytvořen automat odpovídající sjednocení obou regulárních výrazů a je rovněž determinizován a minimalizován. Na standartní výstup potom funkce vypíše dvojici regulárních výrazů a za nimi desetinné číslo odpovídající koeficientu interakce (vypočítané podle vzorce v sekci 4.2). V tomto případě není využita vlastnost modelu MapReduce, která dovoluje rozdělit výstup podle různých klíčů, protože funcke Reduce() potřebuje kompletní data ze všech mapperů a nezáleží na pořadí, v jakém data přijme.

Reducer načte postupně celý svůj vstup a ukládá data do grafové struktury, jak je naznačeno v algoritmu 1. Původně jsem se snažil najít vhodnou Python knihovnu, která umí pracovat s grafy, nicméně jsem nenašel žádnou takovou, která by vyhovovala mému využití (důraz na rychlost přístupu k ohodnocení hrany mezi dvěma uzly, případně na počet hran vycházejících z uzlu). Proto jsem se nakonec rozhodl použít dvojrozměrnou strukturu Python slovníků (dictionary), kde jsou klíčem čísla reprezentující regulární výraz. Pro každou dvojici regulárních výrazů re1 a re2 je koeficient uložen v grafové strukuře g do slovníku v g[re1] pod indexem re2 a současně do slovníku v g[re2] pod indexem re1. Při výběru mezi slovníkem a seznamem v Pythonu jsem dal přednost slovníku, ačkoliv má sice znatelně vyšší paměťové nároky (experimentálně jsem naměřil cca trojnásobnou paměťovou náročnost) a budování datové struktury trvá déle, ale náročnost na procesor při přístupu je díky využití hashovací tabulky nižší než u seznamu, navíc budování struktury probíhá pouze na začátku a pak už program do struktury především přistupuje a mění pouze její obsah.

Po načtení celého vstupu do grafu začne reducer v souladu s vybraným algoritmem provádět shlukování regulárních výrazů. Následuje hlavní cyklus funkce Reduce, kde je nejprve vybrán uzel grafu, ze kterého vychází nejnižší počet hran (případně pro algoritmus 2 najde hranu s nejnižším ohodnocením) a provede determinizaci prvního RV. Poté provádí vnořený cyklus, který hledá vhodné kandidáty na sjednocování. Nejprve jsou ale uloženy informace o aktualním shluku do pomocné proměnné, aby bylo možné v případě, že nově vytvořený shluk překročí stanovený limit pro počet stavů konečného automatu, přerušit vnořený cyklus a vytvořit nový shluk. Dále se od sebe jednotlivé algoritmy liší. První algoritmus pouze prochází datovou strukturu a sčítá počty hran mezi jednotlivými uzly grafu a nově vytvořenou skupinou. Druhý algoritmus provádí pro každý zbývající RV v grafové struktuře výpočet koeficientu interakce vůči sjednocenému konečnému automatu aktuálního shluku. Oba poté podle nejlepší hodnoty vyberou RV (ten je zároveň odstraněn z celé skupiny), vytvoří sjednocení a po determinizaci a minimalizaci ověří, jestli počet stavů automatu nepřekročil limit. Pokud ho překročí, je ukončen vnořený cyklus a z pomocné proměnné je vrácena aktuální skupina. V opačném případě se cyklus opakuje. Pokud již v grafové struktuře není žádný uzel, je ukončen hlavní cyklus a funkce reduce vrací jednotlivé automaty, případně i regulární výrazy, ze kterých se skládají.

Při implementaci jsem ještě provedl optimalizaci algoritmu 1. Místo výběru vhodného regulárního výrazu pro shlukování podle počtu hran vedoucích do uzlů nového shluku ná-

stroj vytvoři grafovou strukturu včetně koeficientů interakce a hledá regulární výraz, který má nejnižší součet těchto koeficientů s regulárními výrazy z vytvářeného shluku.

6.2.2 Více MapReduce úloh

Návrh s více MapReduce úlohami již vyžaduje částečné odchýlení od navrhovaných shlukovacích algoritmů. To především kvůli škálovatelnosti výpočtu. V tomto návrhu probíhá funkce map úplně stejně jako v předchozím návrhu, funkce Reduce už ale funguje jen k posbírání výsledků a ve frameworku Hadoop je tedy možné jí uplně vynechat. Výstupem prvni MapReduce úlohy je tedy seznam dvojic regulárních výrazů s desetinným číslem reprezentujícím koeficient interakce.

Po proběhnutí první MapRededuce úlohy je třeba rozdělit výstup pro mappery další MapReduce úlohy. Původně jsem vstup rozděloval následujícím způsobem: výstup první MapReduce úlohy jsem celý načetl do skriptu, který vypočítal pro každý regulární výraz průměrný koeficient interakce a seřadil celý seznam vzestupně podle vypočteného průměru R. Tento seznam jsem poté na vstup další MapReduce úlohy rozděloval po konstantních počtech řádků do souborů, tedy například po 100 regulárních výrazech. Tento způsob se ale nakonec úkázal být neefektivní. Některé mapovací úlohy druhé MapReduce úlohy totiž trvaly neúměrně dlouho (několikanásobně déle než ostatní), protože obsahovaly velký počet výrazů, které způsobují „explozi stavů“ (viz kapitola o slukování 4), případně regulární výrazy s velkým počtem stavů a tím pádem determinizace trvala dlouho.

Došel jsem tedy k následujícímu řešení: Skript splitREList.py nejprve zadá krátkou MapReduce úlohu, která každý regulární výraz převede na deterministický konečný automat a vrátí počet jeho stavů N. Poté si skript stanoví hraniční hodnotu podle limitu pro počet stavů shlukovaných regulárních výrazů (například 3-4 násobek limitu počtu stavů - tuto hodnotu by si teoreticky mohl nastavovat uživatel sám a tím i určit poměr mezi efektivitou shlukování a časovou náročností), následně načítá postupně řádky vygenerovaného seznamu a sčítá hodnotu vypočítanou jako $N * R_{NG}$, kde R_{NG} je průměrný KI vůči regulárním výrazům, které již ve skupině jsou a N je počet stavů RV. Při překročení hraniční hodnoty vytvoří skript blok pro následující MapReduce úlohu a postup opakuje, dokud nerozdělí celý seznam. Každý z těchto bloků je postupně zapsán do svého souboru.

Shlukování regulárních výrazů u tohoto řešení probíha ve funcki Map() další MapReduce úlohy. Ta má přístup k datům z první MapReduce úlohy, takže již nemusí přepočítávat KI a stejným způsobem jako u předchozí implementace shlukuje RV. Rozdíl nastává při vyčerpání všech výrazů v seznamu. Může se stát, že poslední vznikne například pouze z jednoho RV. Proto pokud počet stavů konečného automatu, který odpovídá poslednímu shluku, nepřekročí alespoň 90 procent limitu, vrátí mapper místo tohoto shluku regulární výrazy které do něj patří. Hodnota 90 procent nemá žádný podklad, ale vychází spíše z pozorování výsledků při testování aplikace a mohla by teoreticky být nastavitelná uživatelem. Tyto „zbytky“ nakonec posbírá reducer a ten buď nad všemi nakonec provede ještě sám shlukování, případně pokud zbude regulárních výrazů příliš mnoho, předá neshlukované výrazy hlavnímu skriptu a ten opět RV seřadí a opakuje postup popsaný výše.

6.2.3 Velké množství krátkých MapReduce úloh

Další možností, jak využít cluster pro shlukování, je v kombinaci s algoritmem 2, kde je nutno přepočítávat KI po každém výběru RV. Při přepočítávání shluku složeného ze dvou RV trvá přepočítání relativně krátkou dobu, ale když už se shluk blíží limitu stavů pro

výstupní automat, může trvat přegenerování grafové struktury po jednom kroku řádově 10x-100x více v kapitole 7. Vytvořil jsem tedy verzi shlukování, kde je možné využít Hadoop k relativně rychlému přepočítání grafové struktury. Přepočítávání grafové struktury pro jeden krok se i pro měnší množství regulárních výrazů (např. 300) na vstupu někdy dostalo až na stovky sekund při výpočtech na jediném počítači. Jednotlivé výpočty jsou na sobě nezávislé, takže se hodí pro distribuované výpočty. Program tedy zapíše do souboru na každý řádek zvlášť vždy serializovaný konečný automat aktuálního shluku a za ním oddělený tabulátorem regulární výraz. Funkce Map() v této MapReduce úloze bude provádět stejný výpočet jako u předchozích návrhů a bude mít i stejný výstup. Funkce Reduce() bude pouze shromažďovat výstupy Mapperů. Po dokončení MapReduce úlohy si hlavní program načte soubor s výstupem a pokračuje výběrem vhodného RV do shluku.

6.3 Vylepšení metod shlukování

V rámci této práce jsem také zkoumal, jak optimalizovat algoritmy, které již byly navrženy dříve. A to jak pro klasický přístup, tak i pro přístup pomocí modelu MapReduce.

6.3.1 Vylepšení algoritmu popsaného v 4.1

Pro algoritmus 1 jsem zkoušel různá vylepšení pro výběr nejvhodnějšího regulárního výrazu. Během implementace a testování tohoto algoritmu jsem zjistil, že na konci celého procesu shlukování často zůstavají regulární výrazy, které mají obecně velkou míru interakce. Proto se poslední shluky tvoří pouze z několika regulárních výrazů, často dochází k expanzi stavů a navíc nebývá využita plná kapacita limitu stavů. Zkoušel jsem tedy způsob dvou průchodů, kdy první průchod provede klasické shlukování a regulární výrazy, které zůstaly na konci jsou předřazeny na začátek a využity vždy jako první do nové skupiny. Tím by měla být vyplněna mezera v posledních shlucích, nicméně výsledky testů ukázaly, že tento přístup shlukování nezefektivní, ale naopak efektivitu snižuje.

Další způsob jak vylepšit tento algoritmus je využití vzorce pro koeficient interakce definovaný pro inkrementální algoritmus. Hrany jsou v grafové struktuře mezi všemi dvojicemi uzlů a jsou ohodnoceny koeficientem interakce. Výběr regulárního výrazu do shluku není prováděn podle počtu hran vedoucích k uzlům ve vytvářené skupině, ale podle součtu koeficientů interakce těchto hran. Tento způsob již přináší určité zefektivnění shlukování s relativně minimálním nárůstem procesorové a paměťové náročnosti. Porovnání efektivity a časové náročnosti oproti původnímu algoritmu je v kapitole 7.

6.3.2 Vylepšení inkrementálního algoritmu

Inkrementální algoritmus je oproti algoritmu 1 účinnější z hlediska shlukování, ale jeho časová náročnost může značně prodloužit shlukování a to i s využitím Hadoopu. Proto jsem zkoušel různé metody jak urychlit shlukování. Jednou z nich je již zmíněné využití krátkých MapReduce úloh, které zkrátí přepočítávání koeficientů interakce při zachování efektivity algoritmu.

Dalšími možnostmi, které jsem zkoumal byla predikce regulárního výrazu, který algoritmus vybere v dalším kroku. Predikované výrazy byly vybírány podle průměrného KI vůči těm, které již ve skupině jsou a zároveň podle posledního vypočítaného KI vůči celému shluku. Dále jsem stanovil 2 hranice, maximální a minimální počet regulárních výrazů, které budou v seznamu odhadovaných RV. Pokud jsou hranice nastaveny např. na 10 a 15

RV, algoritmus na začátku přepočítá celý seznam. Po výběru prvního RV do shluku vybere program podle zmíněných hodnot 15 RV, mezi kterými je pravděpodobně ten, který by byl vybrán v dalším kroku. Místo přepočítávání celé grafové struktury následně přepočítává aplikace pouze KI pro predikované RV vůči vytvářenému shluku a ostatní uzly při výběru RV do shluku ignoruje. Takto pokračuje, dokud počet výrazů v predikované skupině není roven 10 (minimální hranice). Poté přepočítá celou grafovou strukturu a znovu vytvoří predikovanou skupinu. Díky tomuto přístupu lze proces shlukování zrychlit, ale již za částečného snížení účinnosti. Při experimentálních testech jsem naměřil, že pravděpodobnost, že regulární výraz, který by algoritmus vybral v dalším kroku se nachází v predikované skupině se pohybuje podle nastavení horní a dolní hranice mezi 60ti a 90ti procenty.

O něco jednodušší způsob jak zkrátit přepočítávání ohodnocení hran v grafové struktuře je vypočítat průměrnou hodnotu KI mezi vytvářeným shlukem a všemi ostatními RV a aktualizovat pouze ty hrany, které mají ohodnocení nižší než vypočítaný průměr. Tato metoda také zkrátí shlukování, ale testováním jsem došel k závěru, že má značně větší dopad na efektivitu shlukování.

6.4 Parametry programu a použití

V této sekci jsou zmíněny hlavní parametry pro skript a je popsáno použití aplikace. Ostatní volby a přepínače jsou popsány v souboru README. Pro využití Hadoopu je nutné mít framework nainstalovaný na celém clusteru a zároveň zajistit, aby byl nástroj NetBench a všechny potřebné Python knihovny přístupné v příslušných adresářích a spustitelné na všech počítačích v clusteru. Zároveň musí být všude nastavena systémová konstanta NETBENCHPATH pro použití nástroje NetBench a konstanta HADOOPBINPATH pro použití Hadoopu (viz soubor README). Hadoop zároveň musí být spuštěn.

6.4.1 Výběr algoritmu pro shlukování

Základní volbou pro uživatele je výběr algoritmu, který bude provádět shlukování. Implicitně používá aplikace algoritmus 1 s vylepšením ze sekce 6.3.1. Dále může být vybrán algoritmus 1 bez vylepšení, inkerementální algoritmus a přístup popsaný v sekci 4.3.

6.4.2 Shlukování s použitím Hadoop

Další volbou pro uživatele je, zda chce využívat model MapReduce, nebo ne. Implicitně je nastaveno použití Hadoopu a v takovém případě je framework využíván všude, kde to umožňuje návrh aplikace.

6.4.3 Stanovení maximálního počtu stavů KA po shlukování

Dále si uživatel může zvolit limit počtu stavů ve výsledných shlucích. K tomu slouží parametr -s [hodnota]. Pokud uživatel hodnotu nenastaví, je použita hodnota 500. Tento parametr je důležitý především pro následné využití shluků, například kvůli paměťovým omezením zařízení, na kterém budou výsledné automaty využívány (například L1 Cache u procesoru). Výše limitu je individuální pro každého uživatele a proto se v rámci této práce nezabývám určováním ideální hodnoty. Implicitní limit 500 je pouze orientační.

Kapitola 7

Testy a hodnocení

Pro porovnání rychlosti celého procesu shlukování regulárních výrazů s použitím clusteru a bez něj jsem využil cluster sedmi počítačů, ke kterým jsem měl přístup (v testech bude tento cluster označen jako CL1). Bohužel výkonnost těchto počítačů byla nevyvážená a často docházelo k chybám. Proto jsem se nakonec rozhodl vyzkoušet navíc ještě Amazon Elastic Compute Cloud (EC2) [1], který v testech bude označen jako CL2 a CL3. EC2 je webová služba poskytující pronájem serverů a nad nimi mimo jiné i vlastní implementaci modelu MapReduce zvanou Elastic MapReduce [2] (EMR). EMR vychází z implementace Apache Hadoop. Pro porovnání s klasickým přístupem jsem původně chtěl využít školní server eva. Nakonec jsem se ale rozhodl využít vlastní počítač (v testech označen jako PC1), protože většina úloh proběhla rychleji na mém počítači než na školním serveru. Testování jsem rozdělil do několika podčástí, především protože na sobě nejsou závislé a poskytnou případnému uživateli lepší přehled o efektivitě a časové náročnosti jednotlivých možností.

Nejprve je popsána specifikace sestav pro CL1,CL2 a PC1. Dále sekce 7.2 popisuje testování rychlosti vytváření grafové struktury a výpočtu koeficientu interakce pro všechny dvojice regulárních výrazů ze vstupního seznamu. V sekci 7.3 následuje porovnání rychlosti a efektivity shlukování mezi algoritmy popsanými v kapitole 5. Sekce 7.4 porovnává rychlost a efektivitu algoritmů při využití distribuovaných výpočtů oproti klasickému přístupu. V sekcích 7.5 a 7.6 je porovnání všech kombinací a doporučení výběru pro uživatele a hodnocení celkového využití modelu MapReduce a frameworku Apache Hadoop pro shlukování regulárních výrazů.

7.1 Specifikace testovacích sestav a vytížení jednotlivých komponent

Již po krátkém testování bylo zřejmé, že celý proces shlukování bude především náročný na procesor, proto se ve specifikacích budu soustředit především na výpočetní jednotku. Paměťové nároky na shlukování jsou vzhledem k současným konfiguracím strojů zcela zanedbatelné. Pouze u velkého množství regulárních výrazů může nastat problém v části vytváření grafové struktury. U algoritmu 2 je paměťová náročnost grafové struktury pro 100 regulárních výrazů cca 600 KB, 10 000 regulárních výrazů na vstupu už si vyžádá téměř 4 GB paměti. Již z principu ukládání těchto dat (dvojrozměrná struktura slovníků) je zřejmé, že paměťová náročnost grafové struktury bude narůstat kvadraticky vzhledem k

[1] <http://aws.amazon.com/ec2/>

[2] <https://aws.amazon.com/elasticmapreduce/>

počtu regulárních výrazů. Pokud by uživatel vyžadoval snížení paměťových nároků, bylo by možné zvolit strukturu dvojrozměrného pole. Tím se paměťová náročnost sníží cca na třetinu.

Cluster označený jako CL1 se skládal z celkem sedmi počítačů s operačním systémem Ubuntu 12.04 a byl propojený v gigabitové síti. Tři z počítačů měly shodnou konfiguraci, konkrétně dvoujádrové Intel Pentium Dual-Core s taktem 2 Ghz (Jeden z nich byl podtaktovaný na 1,6 Ghz) a dále 2 GB paměti. Dva počítače fungovaly s čtyřjádrovým procesorem Intel Core i5 s taktem 2,9 Ghz a 4 GB paměti. Procesor v posledním počítači byl jednojádrový Intel Celeron s taktem 1,6 Ghz a 2 GB paměti. Bohužel u jednoho ze dvou core i5 procesorů docházelo k přehřívání a k chybám. Apache Hadoop se sice z těchto potíží dokázal bez problémů zotavit, nicméně tento počítač představoval cca čtvrtinu výpočetní kapacity celého clusteru a tato skutečnost zřejmě ovlivnila výsledky zejména u déle trvajících testů.

Clustery CL2 a CL3 byly složeny ze strojů m1.medium z nabídky Amazon EC2 [3]. CL2 z 20ti strojů a CL3 z 10ti strojů. Každý z těchto strojů má k dispozici jednojádrový procesor s taktem 2-2,4 Ghz a 3,75 GB paměti. Vstupní data a zdrojové kódy v tomto případě bere Elastic MapReduce z cloudového úložiště Amazon S3 [4].

Počítač, na kterém jsem testoval klasický přístup, má procesor Intel Core i5 s taktem 2,83 Ghz a 8 GB paměti.

Veškeré výpočty byly prováděny bez jakékoliv paralelizace na úrovni kódu v Pythonu, takže u klasického přístupu bylo zatěžováno pouze jedno jádro a to v průměru na 95% u většiny úloh. Pro plné využití výpočetní kapacity clusteru jsem nastavoval ve frameworku Hadoop dva výpočetní sloty na jedno jádro procesoru, to znamená dva procesy na každé jádro. Díky tomu bylo vytížení procesorů u počítačů v clusteru po většinu doby kolem 96%, s jedinou vyjímkou a tou byl hlavní uzel (master), u něj se vytížení pohybovalo v průměru kolem 70%. Během výpočtu jsem také měřil vytížení síťového provozu u master uzlu pomocí nástroje ifstat. Využití sítě nepřesáhlo 40% a průměrná hodnota se pohybovala kolem 2%. Stejně tak paměťové nároky se pohybovaly kromě výše zmíněné části do 50-100MB pro většinu testů.

7.2 Vytváření grafové struktury a výpočet koeficientu interakce

Výpočet koeficientu interakce pro všechny dvojice regulárních výrazů je časově nejnáročnější část výpočtů, především kvůli determinizaci konečných automatů. Zároveň ale jednotlivé výpočty na sobě nejsou nijak závislé a nevyžadují ani přenos velkých množství dat. Díky těmto skutečnostem dokáže distribuovaný výpočet značně zrychlit shlukování regulárních výrazů. Počet determinizací roste kvadraticky vůčí počtu vstupních regulárních výrazů. Pro názornou ukázku časové náročnosti je porovnána doba trvání postupně pro různé hodnoty od 10 do 1000 regulárních výrazů. Podrobné výsledky jsou znázorněny v tabulce 7.1.

Jak je možné vidět v tabulce 7.1, pro menší počty regulárních výrazů je rychlejší klasický přístup. To je způsobeno především relativně velkou režií při spouštění MapReduce úlohy (viz kapitola 3). Pro vyšší počty regulárních výrazů (50 a více) je již rychlejší využití clusteru. Pro 1000 a více regulárních výrazů se už stal klasický přístup v zásadě nepoužitelný, protože výpočet trval přes 56 hodin, kdežto cluster úlohu zvládl za 6,5 hodiny. Navíc by

[3] <https://aws.amazon.com/ec2/instance-types/>

[4] <http://aws.amazon.com/s3/>

Počet RV	Počet determinizací	Čas na PC1	Čas na CL1	Čas na CL2	Čas na CL3
10	55	4s	20s	34s	33s
20	210	21s	23s	35s	34s
50	1275	5m 25s	3m 44s	3m 1s	3m 59s
100	5050	19m	6m 12s	5m 7s	7m
200	20100	2h 23m	35m	29m	33m
500	125250	7h 12m	1h 25m	57m	2h 2m
1000	500500	56h 40m	6h 31m	3h 59m	8h 27m
2000	2001000	X	33h 14m	X	X

Tabulka 7.1: Porovnání rychlosti výpočtů koeficientu interakce pro všechny dvojice regulárních výrazů ze vstupu. (testy s výsledkem X nebyly provedeny)

bylo možné tento čas ještě výrazně zkrátit zvýšením počtu výpočetních jednotek (poměr časů CL2 a CL3 přibližně odpovídá poměru počtů strojů v clusteru).

Podle dat v tomto tabulce 7.1 se zdá, že se průměrná časová náročnost determinizace zvyšuje úměrně k počtu regulárních výrazů, nicméně tato odchylka je spíše způsobená tím, že větší vstupní seznamy již obsahovaly větší množství výrazů, které způsobovaly explozi stavů. Příkladem je např regulární výraz $/T.*?T.*?Y.*?P.*?R.*?O.*?M.*?P.*?T/RBi$ ze Snortu který u testovacího vzorku s 1000 RV měl průměrný KI roven 4.286115007 (v průměru při sjednocení s jiným RV měl výsledný automat 4x více stavů než oba původni dohromady) a průměrný čas výpočtu KI byl roven $3.54s$, což je téměř sedminásobek průměrného času pro všechny RV. Takové regulární výrazy by bylo vhodné odhalit již při zpracovávání vstupu a nevypočítávat pro ně zbytečně KI.

Jak je zřejmé ze statistik, tato část výpočtu je závislá především na výpočetní síle HW na kterém jsou výpočty prováděny. Zároveň jediná uživatelská data přenášená po síti jsou vstupní dvojice regulárních výrazů a výstupní dvojice doplněné o koeficient interakce, vytížení sítě je tedy vzhledem k počtu regulárních výrazů minimální. Díky těmto skutečnostem můžeme pro vytváření dat pro grafovou strukturu plně využít potenciál distribuovaných výpočtů. Horní hranicí pro škálovatelnost by tedy nejspíše bylo technologické omezení frameworku Apache Hadoop.

7.3 Porovnání rychlosti a efektivity shlukovacích algoritmů

Porovnání algoritmů 1 a 2 je již popsáno v článku [1], v rámci této práce jsem se ale rozhodl tyto testy zopakovat, pokusit se potvrdit výsledky a zároveň porovnat časovou náročnost a efektivitu obou přístupů oproti mnou navrženým změnám a přístupům. Již předem bylo jisté, že shlukování pomocí inkrementálního algoritmu bude z časového hlediska nesrovnatelně náročnější oproti algoritmu 1. V tabulce 7.4 jsou výsledky testů časové náročnosti (s různými limity stavů výsledných konečných automatů) pro algoritmy popsané v kapitole 5.

Kromě inkrementálního algoritmu probíhalo shlukování vesměs stejnou dobu se vzájemnou odchylkou do 10%. Navíc kvůli vysoké časové náročnosti shlukování jsem většinu testů prováděl pouze jednou, a proto nemá přesné porovnávání rychlosti jednotlivých přístupů smysl. Jako nejúčinnější, ale zároveň nejpomalejší se úkázal podle předpokladů inkrementální algoritmus. Porovnání časové náročnosti pokračuje v další kapitole.

Počet RV	Limit stavů	Algoritmus 1 [počet stavů]	Algoritmus 1 s vylepšením [počet stavů]	Algoritmus 2 [počet stavů]	Algoritmus 4.3 [počet stavů]
200	300	1 489	1 386	1 377	1 501
200	500	1 750	1 531	1 570	1 649
400	300	10 254	9150	8946	11 843
400	500	12 168	11 319	11 102	14 002
400	1000	16 038	14 638	x	18 756
700	300	17 706	15 801	14 689	19 830
700	500	19 938	18 900	18 498	22 523
700	1000	27 292	24 287	x	30 340

Tabulka 7.2: Porovnání efektivity shlukovacích algoritmů z hlediska součtu stavů výsledných shluků (pokud je v tabulce X, test nebyl proveden)

Počet RV	Limit stavů	Algoritmus 1 [počet shluků]	Algoritmus 1 s vylepšením [počet shluků]	Algoritmus 2 [počet shluků]	Algoritmus 4.3 [počet shluků]
200	300	7	6	6	7
200	500	6	5	5	5
400	300	44	43	41	50
400	500	31	31	30	35
400	1000	23	22	x	26
700	300	77	73	68	85
700	500	56	53	52	64
700	1000	37	35	x	40

Tabulka 7.3: Porovnání efektivity shlukovacích algoritmů z hlediska počtu shluků (pokud je v tabulce X, test nebyl proveden)

7.4 Výběr regulárních výrazů do shluku s a bez pomoci clusteru

Kromě inkrementálního algoritmu probíhají všechny shlukovací metody relativně rychle a řešení s více MapReduce úlohami navrhované v 6.2.2 sice zrychlí jejich provádění, ale sníží efektivitu shlukování. Při experimentech s různými hraničními hodnotami trvalo shlukování v průměru 5-8krát kratší dobu (bez možnosti dalšího zrychlení zvětšením clusteru), ale vzniklých shluků bylo cca o 15-30% více, přičemž platilo čím větší urychlení, tím menší efektivita shlukování. Aby bylo využití MapReduce pro shlukování přístupem 6.2.2 efektivnější, bylo by nutné vylepšit algoritmus pro rozdělování RV do skupin pro jednotlivé shlukovače.

Z výsledků experimentů jsem usoudil, že z mnou navrhnutých řešení se distribuovaný výpočet vyplatí využít jedině u inkrementálního algoritmu, s použitím návrhu 6.2.3. Porovnání rychlostí provádění těchto algoritmů s clusterem a bez něj jsem provedl jen pro malé počty regulárních výrazů a pro nízké limity stavů automatu odpovídajícímu vytvořenému shluku. Již u 200 RV s limitem 300 stavů trval například první krok výběru RV do shluku 10s. Poslední krok před vytvořením druhého shluku již trval 450s. Celkový čas shlukování

bez použití clusteru překročil 7 hodin. Pro 200 RV s limitem 500 stavů trval celý proces 23 hodin 36 minut, takže jsem se rozhodl další testy provádět již jen s využitím clusteru.

Při využití clusteru pro vytváření shluků inkrementálním algoritmem již byl proces značně rychlejší, i když stále o dost pomalejší než ostatní algoritmy. Důležité ovšem je, že tento proces shlukování je možné zrychlit zvětšením clusteru, a to teoreticky až po hranici, kdy by bylo v clusteru stejně strojů jako regulárních výrazů na vstupu. Porovnání rychlostí shlukování na clusteru oproti PC1 a oproti algoritmu 1 je v tabulce 7.4.

Počet RV	Limit stavů	Algoritmus 2 na PC1	Algoritmus 2 na CL1	Algoritmus 1
200	300	7h 4m	55m	10m
200	500	23h 36m	4h 47m	14m
700	300	X	6h 51m	40m
700	500	X	13h 47m	1h 14m

Tabulka 7.4: Porovnání efektivity shlukovacích algoritmů (pokud je v tabulce X, test nebyl proveden)

7.5 Využití modelu MapReduce pro shlukování regulárních výrazů

Naměřené hodnoty v této kapitole ukazují, že model MapReduce a jeho implementace v rámci frameworku Hadoop dokáží výrazně zrychlit shlukování regulárních výrazů. Především dokáží zkrátit úvodní fázi shlukování, ve které jsou připravována data pro grafovou strukturu. Míra zrychlení této fáze je úměrná výpočetní síle clusteru, na kterém jsou výpočty prováděny, tzn. tato část výpočtu je škálovatelná.

V části výpočtu kde dochází k výběru RV a vytváření výstupních shluků jsem nebyl schopen pro algoritmus 1 najít účinný postup, který by byl schopen výrazně zrychlit shlukování, při zachování efektivity oproti klasickému přístupu. Při použití tohoto algoritmu by tedy bylo výhodnější po vytvoření grafové struktury využít spíše jeden výkonnější server oproti clusteru, navíc jak je vidět v tabulce 7.4 druhá část výpočtu u algoritmu 1 není zdaleka tak náročná jako u algoritmu 2.

U algoritmu 2 už ale můžeme docílit zrychlení způsobem popsaným v 6.2.3 při zachování stejné efektivity jako u klasického přistupu. I když tento algoritmus dokáže shlukovat efektivněji než 1, je pro větší množství regulárních výrazů na vstupu (500 a více) téměř nepoužitelný, protože přepočítávání KI může prodloužit shlukování o desítky hodin.

Po rozdělení do skupin RV jak je popsáno v 6.2.2 je možné využít na jednotlivé skupiny kterýkoliv ze tří přístupů popsaných v kapitole 4. Díky využití distribuovaných výpočtů dokáže tento způsob přinést značné zrychlení, ale má příliš velký vliv na efektivitu shlukování.

7.6 Doporučení pro uživatele

Pro uživatele, který potřebuje co nejlepší efektivitu shlukování a nezáleží mu na době shlukování bude nejlépe vyhovovat postup navržený v 6.2.3. Tento algoritmus obecně dosahuje nejlepších výsledků při shlukování (viz tabulky 7.3 a 7.2) a díky využití distribuovaných výpočtů může trvat relativně krátkou dobu.

Uživatel, který netrvá na maximální efektivitě vytvváření shluků, případně chce co nejrychlejší provedení, je ideální volba algoritmus 4.1 s vylepšením, které jsem popsal v 6.3.1. Tento přístup je mezi mnou testovanými druhý nejúčinnější a patří k nejrychlejším, díky tomu nabízí nejlepší poměr účinnosti vůči časové náročnosti.

Kapitola 8

Závěr

V rámci této bakalářské práce bylo mým cílem prozkoumat možnosti využití modelu MapReduce a frameworku Apache Hadoop pro urychlení shlukování regulárních výrazů. Dále pak za pomoci získaných znalostí navrhnout a implementovat nástroj, který pomocí shlukovacích algoritmů vytvoří skupiny regulárních výrazů tak, aby počet stavů jim odpovídajících konečných automatů nepřesahoval limit stanovený uživatelem.

Při implementaci tohoto nástroje jsem se soustředil na tři základní parametry aplikace. V první řadě na účinnost shlukování, jinými slovy jak co nejvhodněji poskládat regulární výrazy do skupin, aby těchto skupin bylo co nejméně. Druhým parametrem je časová náročnost celého procesu a třetím škálovatelnost. Abych vyhověl prvním dvěma faktorům, vytvořil jsem několik přístupů k realizaci shlukování. Od algoritmu, který shlukování provede nejrychleji, ale s nižší efektivitou až po velice pomalý, ale o mnoho účinnější postup. Případný uživatel může podle svých priorit tyto dva faktory ovlivnit pomocí kombinace parametrů výstupní aplikace. Škálovatelnost výpočtu je u některých přístupů omezená, většinou z toho důvodu, že aplikace potřebuje ucelenou informaci o všech datech a nelze je proto rozdělit mezi více výpočetních uzlů.

Z výsledků testů je zřejmé, že distribuované výpočty a model MapReduce dokáží výrazně zrychlit shlukování regulárních výrazů a to při zachování efektivity shlukování.

Mimo jiné jsem se v rámci této práce zabýval optimalizací již navržených algoritmů pro shlukování a v této práci jsem navrhnul vlastní postup pro shlukování regulárních výrazů (viz kapitola 4), který z hlediska účinnosti podle výsledků testů sice trochu zaostává za ostatními návrhy, nicméně má výrazně nižší paměťové nároky. Porovnání algoritmů bez mých úprav a s nimi je popsáno v kapitole 7.

Možná rozšíření této práce by mohla zahrnovat porovnání jiných implementací modelu MapReduce oproti Apache Hadoop. Další možností by bylo pokusit se vytvořit algoritmus, který by byl schopen předem rozpoznat podobnosti mezi regulárními výrazy a naznačit bez složitých výpočtů rozdělení do skupin.

Literatura

[1] Antonello, R.; Fernandes, S.; Santos, A.; aj.: Efficient DFA grouping for traffic identification. In *Global Communications Conference (GLOBECOM), 2012 IEEE*, Dec 2012, ISSN 1930-529X, s. 1776–1782, doi:10.1109/GLOCOM.2012.6503372.

[2] Costa, P.; Pasin, M.; Bessani, A.; aj.: Byzantine Fault-Tolerant MapReduce: Faults are Not Just Crashes. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, 2011, s. 32–39, doi:10.1109/CloudCom.2011.15.

[3] Dean, J.; Ghemawat., S.: MapReduce: simplified data processing on large clusters. In *Proceedings of 6th Symposium on Operating Systems Design & Implementation*, 2004.

[4] Hazel, P.: PCRE – Perl-compatible regular expressions – man pages [online]. `http://www.pcre.org/pcre.txt`, 2014 [cit. 2014-05-07].

[5] Shvachko, K.; Kuang, H.; Radia, S.; aj.: The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, Washington, DC, USA: IEEE Computer Society, 2010, ISBN 978-1-4244-7152-2, s. 1–10, doi:10.1109/MSST.2010.5496972. URL <http://dx.doi.org/10.1109/MSST.2010.5496972>

[6] Sidhu, R.; Prasanna, V.: Fast Regular Expression Matching Using FPGAs. In *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, March 2001, s. 227–238.

[7] Tie, Y.; Qiang, X.; Jin, H.: A grouping algorithm based on Regular Expression Similarity for DFA construction. In *Communication Technology (ICCT), 2011 IEEE 13th International Conference on*, Sept 2011, s. 671–674, doi:10.1109/ICCT.2011.6157961.

[8] White, T.: *Hadoop: The Definitive Guide. O'Reilly.* Sebastopol, California, 2009.

[9] WWW stránky: Netbench [online]. `http://merlin.fit.vutbr.cz/ant/netbench/index.html`, 2010 [cit. 2014-05-11].

[10] WWW stránky: Snort::snort-rules [online]. `http://www.snort.org/snort-rules/`, 2014 [cit. 2014-05-05].

[11] WWW stránky: Under the Hood: Hadoop Distributed Filesystem reliability with Namenode and Avatarnode. [online]. `https://www.facebook.com/notes/facebook-engineering/under-the-hood-hadoop-distributed-filesystem-reliability-with-namenode-and-avata/10150888759153920`, 2014 [cit. 2014-05-07].

[12] Yu, F.; Chen, Z.; Diao, Y.; aj.: Fast and memory-efficient regular expression matching for deep packet inspection. In *Architecture for Networking and Communications systems, 2006. ANCS 2006. ACM/IEEE Symposium on*, 2006, s. 93–102.