VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2019 Bc. Denis Rexa

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

VÝPOČETNÍ ÚLOHY PRO ŘEŠENÍ PARALELNÍHO ZPRACOVÁNÍ DAT
COMPUTATIONAL TASKS FOR SOLVING PARALLEL DATA PROCESSING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Denis Rexa

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. Jan Mašek, Ph.D.

BRNO 2019

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**
Ústav telekomunikací

Student: Bc. Denis Rexa ***ID:*** 164383
Ročník: 2 ***Akademický rok:*** 2018/19

NÁZEV TÉMATU:

Výpočetní úlohy pro řešení paralelního zpracování dat

POKYNY PRO VYPRACOVÁNÍ:

Prostudujte technologie Apache Spark a Hadoop a vytvořte výpočetní úlohy do cvičení z předmětu Paralelní zpracování dat, který je vyučován v rámci magisterského programu Telekomunikační a informační technika. Úlohy budou řešit zpracování dat s pomocí technologií Kafka a Cassandra, dále základní operace se streamovanými daty a paralelizaci ukázkové optimalizační úlohy. Součástí práce bude vypracovaný návod k těmto úlohám. Celkem budou vytvořeny 4 úlohy. Dále bude provedeno zátěžové testování těchto úloh na výpočetních serverech a výsledky zhodnoceny.

DOPORUČENÁ LITERATURA:

[1] ESTRADA, Raul. Big data smack: a guide to Apache Spark, Mesos, Akka, Cassandra, and Kafka. ISBN 9781484221747.

[2] HAGER, Georg a Gerhard WELLEIN. Introduction to high performance computing for scientists and engineers. Boca Raton: CRC Press, c2011. Chapman & Hall/CRC computational science series, 7. ISBN 978-1-4398-1-92-4.

Termín zadání: 1.2.2019 ***Termín odevzdání:*** 16.5.2019

Vedoucí práce: Ing. Jan Mašek, Ph.D.
Konzultant:

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

Abstrakt

Cieľom tejto práce bolo vytvoriť štyri laboratórne úlohy pre predmet „Paralelní zpracování dat“, kde si študenti vyskúšajú prácu a možnosti technológie Apache Spark ako platformy na paralelný výpočet. Práca zahŕňa aj základné nastavenie a používanie technológie Apache Kafka či NoSQL databáze Apache Cassandra. Zvyšné dve laboratórne úlohy sa zameriavajú na prácu s Problémom obchodného cestujúceho. Prvá z nich má demonštrovať náročnosť úlohy, kde sa študent bude potýkať s exponenciálnym nárastom zložitosti. Druhá úloha pozostáva optimalizačného algoritmu na riešenie problému v clustery. Tento algoritmus je podrobený výkonnostným meraniam v clustery. Záver práce obsahuje odporúčania pre optimalizáciu ale aj porovnanie behu pri rôznom počte výpočtových zariadení.

Kľúčové slová

Apache Spark, Apache Structured Streaming, Apache Kafka, Apache Cassandra, Problém Obchodného Cestujúceho, paralelný výpočet, cluster

Abstract

The goal of this diploma thesis was to create four laboratory exercises for the subject "Parallel Data Processing", where students will try on the options and capabilities of Apache Spark as a parallel computing platform. The work also includes basic setup and use of Apache Kafka technology and NoSQL Apache Cassandra database. The other two lab assignments focus on working with a Travelling Salesman Problem. The first lab was designed to demonstrate the difficulty of a task where the student will face an exponential increase in complexity. The second task consists of an optimization algorithm to solve the problem in cluster. This algorithm is subjected to performance measurements in clusters. The conclusion of the thesis contains recommendations for optimization as well as comparison of running with different number of computing devices.

Keywords

Apache Spark, Apache Structured Streaming, Apache Kafka, Apache Cassandra, Travelling Salesman Problem, parallel computation, cluster

REXA, Denis. *Výpočetní úlohy pro řešení paralelního zpracování dat.* Brno, 2019. 52 s. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedoucí práce: Ing. Jan Mašek, Ph.D.

Prohlášení autora o původnosti díla

Prohlašuji, že svou diplomovou práci na téma Výpočetní úlohy pro řešení paralelního zpracování dat jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

V Brně dne: 15. května 2019 .....

Denis Rexa

POĎAKOVANIE

Rád by som poďakoval vedúcemu diplomovej práce pánovi Ing. Janu Maškovi, Ph.D. za odborné vedenie, konzultácie, trpezlivosť, podnetné návrhy k práci a ochotu s písaním práce aj počas pracovných ciest do zahraničia.

Brno . . . . . .

. . . . . .

Denis Rexa

OBSAH

ZOZNAM OBRÁZKOV

ÚVOD

Množstvo dát tečúci internetom rok od roku rastie a s tým súvisí aj stále komplexnejšia a náročnejšia analýza dát. Architektúra jedného stroja už nevystačuje na potrebné výpočty v reálnom čase. Preto sa prechádza k architektúre viacerých strojov spojených v clustery, kde je možnosť rozložiť záťaž a spustenie paralelných výpočtov.

Začiatok práce sa venuje aktuálnemu nasadeniu open-source technológie Apache Structured Streaming v praxi za posledné 2 roky. Ďalšie kapitoly približujú jeho prínos pri spracovaní dát v reálnom čase, jeho hlavné alternatívne platformy a hlavné rozdiely voči Apache Spark Streaming. Táto technológia je projektovaná na beh v clustery na paralelné výpočty v reálnom čase. Významnou vlastnosťou Apache Structured Streaming je, že pristupuje k neštruktúrovaným dátam (akým je napríklad text) ako k nekonečnej tabuľke s pevnou schémou.

Ďalej sú v práci rozpísané open-source technológie ako Apache Kafka, ktorá predstavuje medzivrstvu medzi zdrojom dát a aplikáciou, ktorá ich spracuje. Druhá z technológií je Apache Cassandra - ľahko škálovateľná NoSQL databáza s vysokou dostupnosťou. Obe zo spomínaných technológií sa nasadzujú vo významných spoločnostiach po celom svete na spracovanie alebo uloženie dát v enormných objemoch. Obe technológie si budú môcť študenti vyskúšať nastaviť a použiť v laboratórnych úlohách, ktoré sú súčasťou tejto diplomové práce.

Hlavný prínos práce predstavujú štyri laboratórne úlohy do novo vznikajúceho predmetu Paralelné spracovanie dát na Fakulte elektrotechniky a komunikačných technológií, VUT v Brne. V prvej laboratórnej úlohe si študent prakticky vyskúša spustenie a nastavenie systému Apache Kafka a úvodné nastavenie NoSQL databáze Apache Cassandra.

V druhej laboratórnej úlohe má študent za úlohu doplniť, resp. pozmeniť kód programu, ktorý sa napája cez API rozhranie do spoločnosti Twitter a posiela údaje generované užívateľmi tejto sociálnej siete do študentom definovanej inštancie Apache Kafka.

Tretia úloha je zameraná na pochopenie problematiky obchodného cestujúceho. V tejto úlohe študenti majú možnosť pozorovať exponenciálny nárast náročnosti výpočtu s pribúdajúcim počtom miest po trase a náplň tejto úlohy spočíva v opäť v doplnení kódu v jazyku Java.

Posledná laboratórna úloha je zameraná na demonštráciu výkonnostnej výhody pri výpočte v clustery. Algoritmus pochádza z voľne dostupných repozitárov a zadanie pre študenta je doplnenie kódu algoritmu za účelom porozumenia princípu prerozdelenia výpočtu.

Posledná časť práce je zameraná na vlastné testovanie algoritmu Problému obchodného cestujúceho. Kapitola obsahuje tabuľky s hodnotami meraní pre varianty s rozdielnym počtom miest a pri postupnom zapojovaní výpočtových zdrojov. Na základe výsledkov sú v závere kapitoly navrhnuté zmeny na optimalizáciu.

1 SÚČASNÉ VYUŽITIE TECHNOLÓGIE APACHE SPARK STREAMING

Veľké objemy dát, ktoré je potrebné spracovať predstavujú v posledných rokoch problém pre tradičné prístupy analýzy dát. Iba pre predstavu, každú minútu sa na stránku Youtube nahrá 48 hodín záznamu, vytvorí sa približne 571 nových webových stránok, na stránkach spoločnosti Facebook pribudne v priemere 34 722 záznamov o označení „Páči sa mi to“ a denne na túto stránku nahrajú užívatelia dáta v objeme 100 terabajtov [7]. V tejto práci preto priblížim možnosti paralelného procesenia v reálnom čase. Jednou z nich je aj Apache Spark Streaming. Je to open-source technológia, ktorá rozširuje možnosti Spark API a využíva sa vďaka jeho širokej škálovateľnosti, vysokej priepustnosti a tolerancií k chybám na spracovanie veľkých objemov dát v reálnom čase [1].

V nasledujúcich podkapitolách priblížim viaceré konkrétne príklady nasadenia, ktoré sa vo svete vyskytli v priebehu posledných pár rokov.

1.1 Monitorovací systém internetu

Podľa štatistík spoločnosti Cisco bolo v roku 2016 internetom prenesených 1,2 zetabajtu (ZB) dát, čo je v numerickom vyjadrení 10^{21} bajtov. Do roku 2021 sa očakáva nárast toku dát naprieč internetom na 3,3 zetabajtu. Tento nárast predstavuje problém pre klasické monitorovacie platformy postavené na architektúre jedného stroja. Aj napriek faktu, že sa používali výkonne stroje tak v prípade veľkého náporu dát, napríklad ako je DDoS útok, sa pakety vzorkovali a to viedlo k nepresnosti výsledkov. Navyše bol systém náchylný na zlyhania a jeho obnova zabrala čas [2].

Pri myšlienke rozloženia záťaže na viaceré stroje v rámci clustera sa najprv uvažovala platforma Hadoop vďaka jeho *MapReduce* modelu, ktorý je veľmi obľúbený v analytickej komunite. Jeho dáta sa však ukladajú na disky, ktoré majú svoje limity v podobe počtu I/O operácií. Pre zvýšenie výkonnosti systému sa ale vybrala technológia Apache Spark. Keďže táto metóda používa *in-memory computing*, teda ukladania a spracovania dát v operačnej pamäti. Dáta sú v tomto prípade rozdelené na tzv. *Resilient Distributed Datasets (RDD).* Klasický prístup spracovávania dát je taký, kedy sú dáta procesené dávkovo. V angličtine tiež známe ako *batch processing*. Čo nie je ale vhodné pri neohraničenom prúde dát prebiehajúci internetom. Knižnice Spark Streaming umožňujú ale techniku tzv. *micro-batching*, kedy sa prúd dát delí na malé sekvencie súborov, ktoré vedia byť rozdelené a spracovávané medzi viacerými výpočtovými uzlami [2].

V práci sa Spark Streaming uplatnil pri meraní výkonnosti a priepustnosti siete, najmä však meranie vlastností TCP pripojení. Implementoval sa paralelný algoritmus na meranie vlastností TCP pripojenia ako oneskorenie *(delay)*, retransmisia, priepustnosť, priemerný *Round-Trip Time (RTT)* a iné. Pri meraní sa vychádzalo z nasledujúcej architektúry:

- *Zberač (Collector)* – zariadenie, ktoré zachytáva všetky prichádzajúce a odchádzajúce rámce na prepínači. V prípade nasadenia na viac prepínačoch je potrebné, aby každý z nich mal vlastný collector.
- *Systém zasielania správ (Messaging system)* – zberač zasiela hlavičky rámcov ako správy, ktoré zachytáva open-source technológia Apache Kafka, ktorá je rozpísaná bližšie v kapitole 3.1.
- *Procesenie záznamu (Stream processor)* – predstavuje Spark Streaming bežiaci v clustery. Informácie získava napojením do Apache Kafka a naslúchaním na konkrétnej téme *(topic)*. Spark Streaming vďaka aplikovaniu viacerých transformačných funkcií ako *flatmap()* či *groupByKey()* dodá požadované informácie. Bližšie popis o funkcií je uvedený v tejto kapitole 2.1 [2].

1.2 WAN: Automatické smerovanie a úprava veľkosti sekvencie

Mnohé typy analýz veľkých dát v reálnom čase sú generované a agregované naprieč WAN sieťami. Teda v prostredí, kedy výpočtové servery nie sú lokalizované v jednej lokálnej sieti a môžu byť fyzicky navzájom oddelené. V prípadoch kedy napríklad operátor dohľadového centra monitoruje systémové logy, aby zistil či nedošlo k chybám a to z jedného miesta pre datacentrá umiestnené po celom svete [3].

Spark Streaming je framework, ktorý je v prípade výpadku primárne zameraný na rýchlu obnovu v rámci jedného datacentra disponujúci vysokorýchlostným pripojením. Pri návrh sa ale nebralo do úvahy rozdielne šírky pásma naprieč pripojenými sieťami, keďže Spark Streaming potrebuje spracovať všetky mikrosekvencie s rovnakou časovou zarážkou *(timestamp)* spolu. Preto prípadné úzke hrdlo po trase môže vyústiť do citeľného zväčšenia odozvy systému. Pre zníženie doby odozvy a dodržanie limitov na spracovanie sekvencií musí systému implementovať nasledujúce kroky [3]:

- vybrať najrýchlejšiu cestu od každého zdroja do siete, kde je umiestnený procesiaci server tak, aby sa pakety vyhli miestam po sieti s menšou šírkou pásma, ktoré by spôsobili oneskorenia. Pri výpočte optimálnej trasy sa teda uprednostňuje šírka pásma pred vzdialenosťou.
- určiť minimálnu dobu trvania sekvencie pre každý dopyt *(query)*, kde sú dáta generované z rôznych zdrojov
- nasadenie algoritmov na medziľahlé uzly, ktoré znížia nadbytočné informácie, ktoré sú posielané naprieč sieťou

Na testovacie účely autori vytvorili 7 inštancií výpočtových zariadení postavených na Amazon EC2 s kontrolovanou šírkou pásma naprieč inštanciami. Na simuláciu úzkeho hrdla, ktoré môže nastať v reálnych WAN sieťach bol nasedený *Linux Traffic Control*. V tomto experimente taktiež porovnávali 2 verzie Spark Streaming systému, ktorý prezentuje odlišné prístupy na výber cesty a veľkosť sekvencie [3]:

- *Wide-area Spark Streaming* - používajú algoritmus popísaný v práci na určenie najvhodnejšej veľkosti sekvencie a optimálnej trasy pre každý dopyt *(query)*
- *Original Spark Streaming* - každý zo zdrojových uzlov posiela jeho lokálne agregované dáta na centralizovaný uzol, kde sú ďalej upravené a dodatočne procesené. V tomto prístupe je taktiež použitý už spomínaný algoritmus na získanie optimálnej veľkosti sekvencie

Pri porovnávaní výsledkov je rozhodujúci čas na spracovanie sekvencií. *Wide-area Spark Streaming* metóda dosahuje lepších časov na sprocesenie pri viacerých typoch dopytov *(queries)* ako štandardne používaná metóda *Original Spark Streaming* avšak rozdiely nie sú zásadne lepšie [3].

1.3 Ďalšie možnosti nasadenia Apache Spark Streaming

Jedným zo zaujímavých nasadení je analýza DNA. Pri každom experimente sú generované dáta v stovkách gigabajtov. Jeden z obľúbených nástrojov na analýzu DNA však nedokáže pracovať v clustery. V tomto článku [4] autori navrhujú framework, tzv. StreamBWA, ktorý dovoľuje distribuovať dáta naprieč clusterom. Týmto spôsobom sa podľa autorov dá skrátiť analýza na približne 50%.

Iné nasadanie Spark Streamingu v článku [5] umožňuje vytvorenie systému na zdieľanie pozície vozidiel, chodcov a udalostí na ceste v reálnom čase komukoľvek, kto používa službu WEWING (asistenčný systém, ktorý spolu s GPS modulom, gyroskopom a akcelerometrom v smartfónoch automaticky deteguje udalosti na cestách ako meškania, nehody, stavy ciest a podobne). Udalosti na ceste sú neskôr zlúčené na základe času výskytu, lokalizácie a obsahu, aby nedošlo k duplikáciám udalostí reportovaných z ciest. V prípade väčšieho počtu užívateľov, ktorí zdieľajú svoje údaje a tieto údaje majú byť v zapätí rozdistribuované naprieč ostatnými užívateľmi v okolí, tak nastáva problém, kedy samotný server so zápisom na disk nie je schopný toto množstvo údajov spracovať v reálnom čase a preto je potrebný výpočtový cluster zariadení.

Ako poslednú ukážku nasadenia je analýza log správ generovaných webovými aplikáciami. Záznamy o návšteve webu či prístupové a chybové logy sú generované vo veľkých objemoch a s vysokou frekvenciou. Preto aj architektúra musí byť robustná a vedieť spracovať dáta v reálnom čase. V práci [6] je popisované nasadenie Spark Streaming na analýzu webových logov reprezentujúce HTTP žiadosti, ktoré prišli na server *NASA Kennedy Space Center*.

2 TECHNOLÓGIA APACHE SPARK

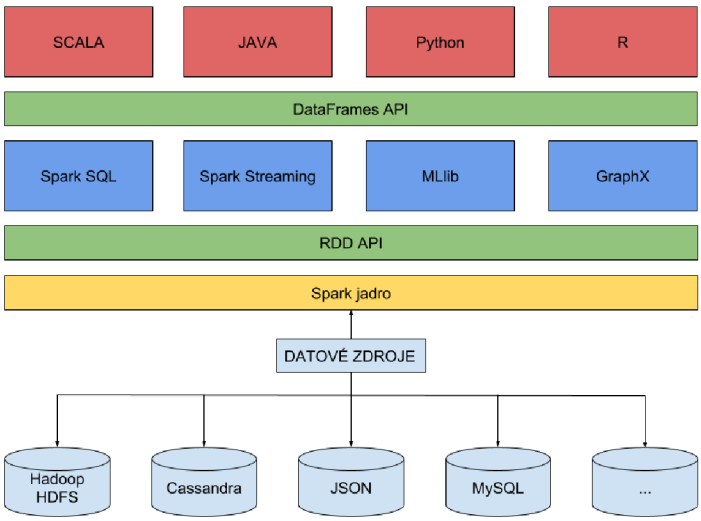
Apache Spark je výpočtový systém pracujúci v clustery, ktorého najväčšie výhody sú rýchlosť a široké možnosti nasadenia. Ponúka API *(Application Programming Interface)* rozhranie v jazykoch Java, Scala, Python či R. V porovnaní s platformou Hadoop ponúka významnú výhodu v podobe rýchlosti načítania dát. Kým Hadoop svoje dáta a priebežné výpočty ukladá na disk, Apache Spark môže pracovať v tzv. *in-memory* režime. To znamená, že na ukladanie využíva operačnú pamäť, ktorá je rýchlejšia. Taktiež Apache Spark poskytuje nástroje ako *Spark SQL* na tvorbu dotazov v jazyku SQL a procesenie štruktúrovaných dát, knižnice *MlLib* pre strojové učenie, *GraphX* pre spracovanie grafových databáz a *Spark Streaming* na spracovanie dát v reálnom čase [8].

Spark SQL, ako modul na spracovanie štruktúrovaných dát, sa využíva na vykonanie SQL príkazov. Môže byť taktiež použitá na čítanie dát z tabuliek *Hive*, ktoré sa využívajú v platforme Apache Hadoop na jednoduchú sumarizáciu dát a analýzu veľkých objemov dát s možnosťou SQL dopytov (queries). Bližšie informácie nájdete na webových stránkach tejto technológie [19].

Knižnica strojového učenia *MlLib* poskytuje algoritmy a nástroje ako [9]:

- Klasifikácia, regresia, zoskupovanie *(clustering)* a kolaboračné filtrovanie
- Extrakcia, transformácia, zníženie dimenzionality a výber
- Konštruovanie, vyhodnocovanie a ladenie ML pipelines
- Lineárna algebra, štatistika a iné

GraphX knižnica je nový modul Sparku pre grafové a grafovo-paralelné výpočty. *GraphX* rozširuje možnosti *Spark RDD (Resilient Distributed Dataset)* novým abstraktným prístupom: orientovaný multigraf s vlastnosťami pripojenými ku každému vrcholu a okraju [10].



Obr. 2.1 Spark architektúra

2.1 Alternatívy k technológií Apache Spark Streaming

V otázke alternatívnych open-source technológií k Apache Spark Streaming závisí na type použitia. V porovnaní s konkurenciou sa Apache Spark stal nepochybne štandardom pri spracovaní veľkých dát.

Medzi najskloňovanejšie alternatívy určite patrí technológia *Apache Storm*. Podobne ako Spark, aj Storm predstavuje framework na spracovanie konštante prúdiacich dát. Bol vytvorený spoločnosťou Twitter a základný rozdiel vychádza zo spôsobu výpočtu. Spark sa spolieha na výpočet tzv. dátového paralelizmu *(data-parallel)*, zatiaľ čo Storm využíva paralelizmus úloh *(task-parallel)* [20]. Rozdiel spočíva v tom, že v prípade paralelného výpočtu úloh sa medzi uzly rozloží výpočet odlišných úloh. No v prípade paralelizmu dát sa medzi výpočtové uzly rozprestrie zátaž výpočtu jednej úlohy.

Ďalšia odlišnosť spočíva v ponúkaných programovacích jazykoch, kedy Apache Spark Streaming ponúka okrem zavedených jazykov ako Java a Scala aj populárne programovacie jazyky rozšírené medzi analytikmi- jazyky Python a R. Posledné dva programovacie jazyky nie sú podporované u technológie Apache Storm [20].

Apache Storm ponúka možnosť dynamickej zmeny topológie bez nutnosti obnovy, resp. prepisu topológie či reštartu celého clusteru. Čo v praxi prináša možnosť pridávať alebo odoberať vykonávateľov *(executors)* alebo výpočtových uzlov *(workers)* počas behu systému [20]. Na druhej strane Apache Spark požaduje, aby bola topológia definovaná v momente, keď sa spúšťa cluster. Prípadná strata uzlu môže viesť k zastaveniu výpočtu [20].

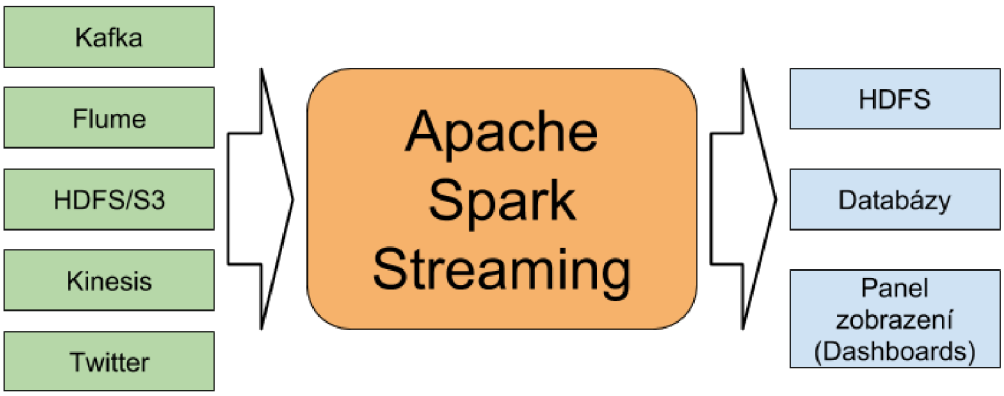
Medzi hlavné alternatívy k Apache Spark Streaming sa taktiež uvádza aj *Apache Flink*. Flink bol od počiatku navrhnutý ako služba na spracovávanie dát v reálnom čase, Spark Streaming je postavený nad jadrom Apache Spark, ktorý nebol vyvíjaný výhradne na spracovanie dát v reálnom čase. Tento rozdiel je vidieť v načítaní dát. Apache Spark pracuje v tzv. *micro-batches*, teda malých blokoch dát nazývaných RDD. Apache Flink pracuje v režime, kedy prúd sa nerozdeľuje ale pracuje s ním súvisle [21]. Hlavným prínosom hovoriacim v prospech Apache Spark je ale rýchlosť. Podľa údajov uvádzaných na stránkach [21] sa Apache Spark Streaming vypláca nasadiť najmä pri väčších objemoch dát. Pri veľkých grafových štruktúrach sa uvádza až 1.7 násobne rýchlejšie spracovanie. V prípade ale menších štruktúr vyhráva Apache Flink vďaka nižšej náročnosti na výpočtové zdroje.

2.2 Apache Spark Streaming

Ako už bolo spomínané vyššie, Apache Spark Streaming umožňuje vytvoriť ľahko škálovateľný cluster s odolnosťou voči chybám, rýchlou opravou a vysokou priepustnosťou. Na obrázku 2 je znázornený tok dát, resp. zdroje dát, ktoré Apache Spark Streaming vie spracovať a systémy, kde tieto informácie vieme posielať. Mimo iné, Spark vie spracovávať ako zdroje dát aj bežné súbory ako JSON (tu je ale treba zadefinovať štruktúru dokumentu) alebo napojenia cez tzv. *socket stream* kde je zdrojom napríklad netcat server. Spark ďalej rozdeľuje príjemcov dát na základe ich spoľahlivosti na dve skupiny [1]:

- *Spoľahlivý príjemca* - posiela správu zdroju o tom, že dáta boli doručené a uložené v poriadku
- *Nespoľahlivý príjemca* - neposiela žiadnu potvrdzovaciu správu zdroju. Môže byť využitý v prípade, kedy zdroj nepodporuje potvrdzovanie alebo aj túto možnosť ponúka, ale potvrdzovacia nie je potrebná

Mimo štandardne podporovaných dátových zdrojov vie Spark prijímať toky dát aj z užívateľom nakonfigurovaných zdrojov. Kedy sa jedná o napojenia napríklad do log záznamov rôznych systémov. Skratka HDFS na obrázku 2.2 označuje *The Hadoop Distributed File System* a jedná sa o distribuovaný súborový systém, ktorý je vhodný pre aplikácie pracujúce s veľkými objemami dát, poskytujúci veľkú priepustnosť a je určený k nasadeniu na hardware s nízkou obstarávacou cenou [11].

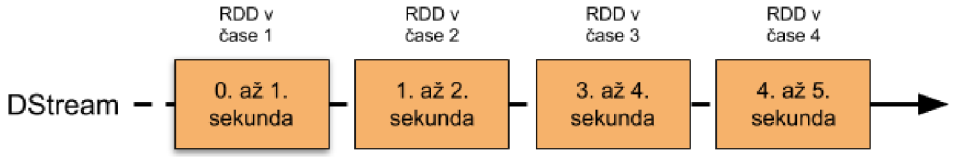


Obr. 2.2 Spark Streaming proces

Spark Streaming funguje v tzv. dávkovom režime, kedy je vstupný tok dát rozdelený na menšie celky *(batches)*, ktoré sú neskôr spracovávané Sparkom. V praxi sa ale používa vyšší level abstrakcie, kedy sa takto rozdelený tok dát obstaráva ako jeden celok, označovaný v angličtine ako *discretized stream* alebo aj *DStream* [1].

Discretized Streams (DStreams)

Každý RDD obsahuje dáta z určitého časového intervalu tak ako to je znázornené na obr. 2.3:



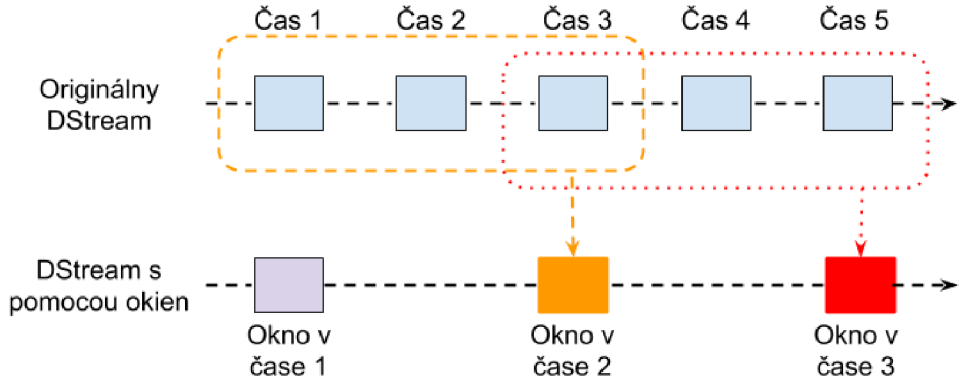
Obr. 2.3 Rozdelenie prúdu dát do RDD

Akákoľvek operácia, ktorá sa aplikuje nad abstraktnou štruktúrou DStream je v pozadí preložená na úkony vykonávané nad jednotlivými RDD. Spark Streaming vie taktiež v jeden okamih spracovávať dáta z viacerých rôznych zdrojov, kde pre každý jeden vstupný tok dát musí existovať odpovedajúci DStream. Dokumentácia Spark Streamingu ďalej ale upozorňuje, že v takom prípade potrebuje Spark alokovaný dostatočný počet jadier, resp. vlákien ak sa jedná o lokálnu inštanciu .

Obdobne ako nad RDD, tak aj v prípade DStreams vieme nad dátami robiť rôzne transformácie a úpravy. Tu je príklad tých bežne používaných s krátkym popisom [1]:

- *Map(func)* - na výstupe tento funkcie je nový DStream, ktorý vznikol úpravou vstupného DStream tak, že každý element na vstupe bol upravený funkciou *func*. V tomto prípade to môže byť akákoľvek funkcia, napr. že každé vstupné slovo prevedie na malé písmená, prenásobí číselnú hodnotu koeficientom a podobne.
- *Filter(func)* - vráti nový DStream s prvkami, ktoré pre funkciu *func* vrátia pravdu
- *Union()* – výsledkom je taktiež nový DStream, kde sa zlúčia 2 DStreams
- *Repartition (numPartitions)* – mení level paralelizmu pridávaním alebo odoberaním partícií *(partitions)*

Pri programovaní vieme taktiež využiť výpočtové okno *(windowed computations)*, kedy sa transformačné funkcie aplikujú na skupinu RRD *(Resilient Distributed Dataset)* za určitý čas. Obrázok 2.4 to približuje:



Obr. 2.4 Znázornenie funkcie okien

Na obrázku číslo 4 je vysvetlená funkcia plávajúceho okna. Vstupné RDD, ktoré spadajú do príslušného okna sú skombinované a počíta sa s nimi ako s jedným celkom. Na obrázku je znázornený prípad, kedy je operácia aplikovaná nad 3 dátovými jednotkami. Toto nastavenie sa označuje ako dĺžka okna *(window length)*. Potom sa okno posunie o 2 diely doprava. V Sparke známe pod nastavením *interval posunutia (sliding interval)* udávajúci dobu, o ktorú sa má okno posunúť v čase [1].

S možnosťou plávajúceho okna sú spojené viaceré možné operácie. Každá z nich ma na vstupe 2 parametre. Tu je ukážka základných z nich [1]:

- *Window (windowLength, slideInterval)* - vráti DStream počítané na základe vstupných parametrov
- *countByWindow (windowLength, slideInterval)* - na výstupe sa objaví počet elementov plávajúceho okna
- *reduceByWindow(func, windowLength, slideInterval)* - vráti prúd dát *(stream)* o jednom elemente, ktorý vznikol agregovaním všetkých elementov v rámci jedného okna za použitia funkcie *func*.

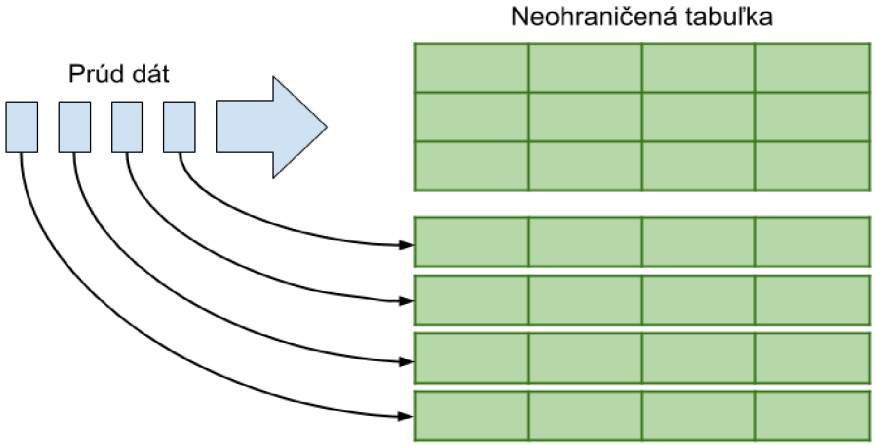
Ďalšia operácia spojená s prúdom dát v Spark Streamingu sú tzv. spájacie operácie *(join operations)*, ktoré poskytujú prepojenie rôznych typov prúdov dát (streams). Poznáme spájanie typu *stream-stream* kedy sú jednoducho zlúčené 2 prúdy dát do jedného za pomoci funkcie *join(stream).* Podobne ako v jazyku SQL, aj v tomto prípade viete určiť typ prepojenia za použitia *leftOuterjoin*, *RightOuterJoin* alebo aj *fullOuterJoin.* Druhá z možností spájania sa označuje ako *stream-dataset*. V tomto prípade sa spája staticky, resp. vopred definovaný dataset a prúd dát. V tomto napojení je však potreba použiť aj funkciu *transform* [1].

Vďaka integrácií dátových rámcov *(DataFrames)* a jazyka SQL je možné pracovať s jednotlivými RDD a použiť nad nimi operácie jazyka SQL. Každý RDD sa skonvertuje na dátový rámec, ktorý sa správa ako dočasná tabuľka *(temporary table)* a umožní aplikácie SQL príkazov. Užívateľ rovnako jednoducho vie aplikovať metódy strojového učenia *(machine learning)* za použitia knižnice *MLlib*. K dispozícií sú algoritmy ako lineárna regresia, Kmeans a podobné [1].

Spark Streaming dovoľuje vývojárom ponechať po určitú dobu dáta v pamäti. Používa sa k tomu metóda *persist()*, ktorá automaticky ponechá každé RDD z daného DStream-u v pamäti. Táto možnosť sa využije v prípade, kedy sa dáta použijú na viaceré výpočty a je nechcené, aby boli zahodené [1].

2.3 Apache Streaming vs. Structured Streaming

Metódou Apache Structured Streaming sa v tejto práci budem detailnejšie zaoberať, keďže som ju uprednostnil pri programovaní pred Apache Spark Streaming. Taktiež disponuje vlastnosťami ako ľahko škálovateľný a odolný voči chybám, no základný rozdiel voči Spark Streamingu vychádza už z jeho názvu - *Structured*, teda štruktúrovaný. Pri súvislom neohraničenom toku dát do systému sa postupne záznamy pridávajú ako riadky do tabuľky. Výpočty sa teda vykonávajú podobne ako pri statických tabuľkách [12].



Obr. 2.5 Postup ukladania dát pri metóde Structured Streaming

Obrázok číslo 2.5 znázorňuje princíp zápisu nových riadkov, kedy sa každý nový záznam pripojí ako riadok na koniec neohraničenej tabuľky (v dokumentácií označený ako *Input Table*, teda vstupná tabuľka). Ako príklad stĺpcov v tejto tabuľke môžu byť hodnota, časová značka, ID generované zdrojom a iné. Funkcia na vstupe (napríklad počítanie konkrétnych slov) potom generuje z dát vo vstupnej tabuľke tzv. *Result Table*. Teda tabuľku, ktorej obsah sa posiela ďalej na výstup. Pri počiatočnom nastavovaní aplikácie sa taktiež definuje v akom móde bude daná aplikácia pracovať. Treba ale myslieť na fakt, že nie všetky typy funkcií podporujú všetky spomínané módy. Poznáme nasledujúce 3 prístupy [12]:

- *Complete mode* - celá aktualizovaná tabuľka je poslaná na výstup. Závisí od napojenia do cieľového úložiska *(storage connector)* ako spracuje celú tabuľku
- *Append mode* - na výstup sa pošlú iba záznamy z *Result Table*, ktoré tam pribudli od posledného zápisu a neočakáva sa pri nich zmena
- *Update mode* - tento mód zachytí iba zmeny *(updates)* v Result Table, ktoré prebehli od posledného zápisu do úložiska

Tento model sa výrazne líši od iných platforiem aj tým, že nevyžaduje od užívateľa aby udržiaval prebiehajúce agregačné procesy, ale Spark nesie zodpovednosť za aktualizovanie *Result Table* v prípade, že na vstup dorazia nové dáta. V prípade, že napríklad chceme robiť analýzu v minútových intervaloch na základe času kedy boli dáta generované, tak sa využije časová pečiatka v *Input Table*. Čo v praxi znamená, že ak nejaký zo záznamov dorazí na vstup

neskôr ako je očakávané na základe časového razítka, Structured Streaming jednoducho prepočíta toto agregačné okno a na výstup pošle informáciu o novej hodnote. Systém však povoľuje definovať časovú zarážku. Teda čas ako dlho má systém ešte udržiavať staré agregačné hodnoty v operačnej pamäti pre prípad oneskorených záznamov. Ak záznam dorazí na vstup po uplynutí tejto zarážky, bude ignorovaný. Táto funkcia je označovaná ako *watermarking* [12].

Vstupné zdroje aktuálne podporované platformou Structured Streaming sú obmedzenejšie ako pri Spark Streamingu. Ale v praxi dosť používaná Kafka samozrejme patrí medzi ne. Mimo Kafky sem patria zdroje ako klasické súbory typu text, JSON, CSV, parquet a UTF-8 text z socket pripojenia, ktorý je momentálne v štádiu testovania. Pri použití klasický súborov je potreba na počiatku definovať ich schému a nespoliehať na Spark, že to vykoná automaticky [12].

Na odstránenie duplikovaných záznamov v dátovom toku sa využíva identifikátor. Pri spracovaní dátového toku sa uložia potrebné dáta na rozpoznanie duplikovaných záznamov a podobne ako pri agregačných funkciách, aj tu môžme využiť metódu časových značiek známu ako *watermarking.* Odlišujeme preto dva rôzne spôsoby pri odstraňovaní duplikačných záznamov [12]:

- *S watermark* - ak máme definované časové obdobie, za ktoré sa duplikované záznamy môžu objaviť, tak vieme využiť stĺpce s časovou značkou a jedinečným identifikátorom v *Input Table* nato, aby sme zmenšili množstvo unikátnych stavov, ktoré systém musí udržiavať a kontrolovať
- *Bez watermark* - keďže neexistuje časová zarážka, za ktorú systém kontroluje možné stavy, systém musí udržiavať kompletný historický zoznam možných stavov

Structured Streaming nepodporuje všetky operácie, ktoré sú dostupné v Spark Streamingu. Tu je zoznam niektorých z nich [12]:

- Viacnásobné, resp. zreťazené agregačné funkcie nad jedným streamovaným datasetom nie sú dostupné
- Distinct operácie nie sú podporované nad streamovaným datasetom
- Zoradenie *(Sorting)* je podporované na streamovaných datasetoch iba po aplikovaní agregačných funkcií
- Žiadne zo spájacích *(join)* funkcií nad dvoma streamovanými datasetami nie je možné

Podobne ako Spark Streaming, aj Structured Streaming ponúka možnosť obnovy po chybe alebo nečakanom zlyhaní. V Spark dokumentácií známe ako *checkpointing*. Pri výstupe môžme jednoducho nastaviť cieľ na fyzickom stroji kde sa informácie o stave behu systému a prebiehajúcich agregačných funkciách budú ukladať. Táto možnosť sa definuje pri výstupnej funkcií *writeStream* zadaním voliteľného parametru *checkpointLocation* [12].

3 UKLADANIE A SPRÁVA DÁT V CLUSTERY

V tejto kapitole priblížim voľne dostupné *(open-source)* technológie na zhromaždenie, resp. ukladania dát v rámci clustera. Jedná sa o technológie Apache Kafka a Apache Cassandra, ktoré boli mnoho krát nasadené vo svete veľkými spoločnosťami. Z výhod Apache Kafka ťažia spoločnosti ako LinkedIn, Yahoo, Twitter, Spotify, Airbnb či Uber napríklad aj na zachytávanie logovacích správ [13]. Apache Cassandra ako NoSQL databáza je nasadená vo firmách ako GitHub, NetFlix či eBay na ukladanie veľkých datasetov [14].

3.1 Apache Kafka

Nejedná sa o databázový systém na ukladanie dát ale skôr nástroj na spracovanie prichádzajúcich tokov dát. Táto platforma sa vyznačuje troma základnými možnosťami nasadenia- vie pracovať ako konzument prichádzajúcich správ, dlhodobo ukladať prichádzajúce záznamy alebo spracovať záznamy ihneď ako prídu na vstup. Bližšie sa budem tomuto rozdeleniu venovať nižšie v tejto podkapitole.

V otázke produkčného nasadenia poznáme dva typy použitia [15]:

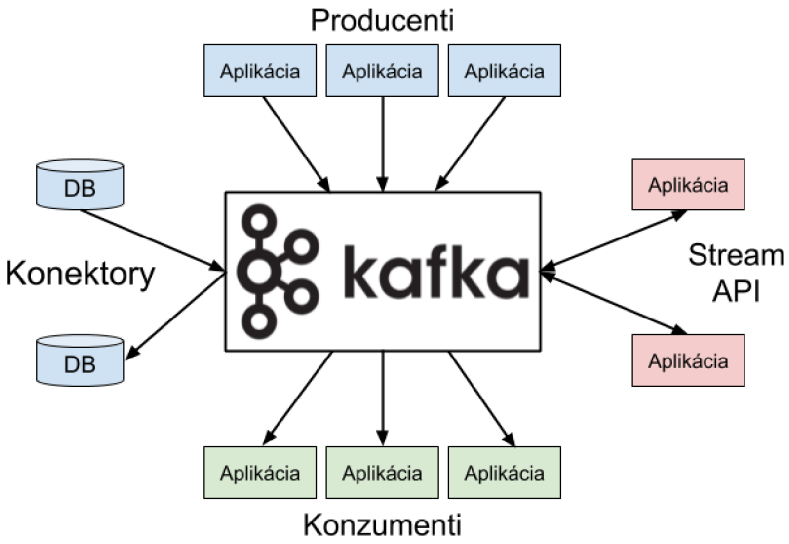
- medzivrstva fungujúca v reálnom čase, ktorá spoľahlivo doručí dáta medzi systémom, ktorý správy generuje a aplikáciou, ktorá ich spracuje
- ako aplikácia, ktorá v reálnom čase transformuje alebo reaguje na prichádzajúci tok dát na vstupe

Technológia Apache Kafka je určená na nasadenie v clustery na jednom či viacerých zariadeniach. Na jednej inštancií sa môžu odchytávať viaceré prichádzajúce zdroje, ktoré sú od seba navzájom odlíšené kategóriami. V dokumentácií sa označujú ako *topics.* Každý záznam disponuje hodnotami ako kľúč, hodnota a časová značka. Týmto spôsobom sa informácie z viacerých aplikácií pôsobiacich na rozdielnych zariadeniach môžu zhromažďovať v jednej kategórií, resp. zložke [15].

Apache Kafka používa na komunikáciu medzi serverom a klientom TCP pripojenie. Ako je znázornené na obrázku 3.1, Apache Kafka rozlišuje tieto štyri typy API napojení [15]:

- *Producenti (Producer API)* - umožňuje aplikáciám aby nahrávali toky dát do jednej či viacero kategórií (*topics*). Producent má na starosti aj samotné rozdelenie záznamov medzi konkrétne partície.
- *Odberatelia (Consumer API)* - aplikácie majú možnosť sa prihlásiť k odberu informácií na jednej či viacero kategórií *(topics)*. Každý odberateľ sa zaradí do určitej skupiny odberateľov a každý záznam pridaný do kategórie sa pošle iba raz na každú z odberateľských skupín. V prípade, že existuje iba jedna odberateľská skupina, záznamy sú rovnomerne rozprestreté medzi konzumentov v rámci skupiny. Naopak, ak existujú skupiny výhradne s jedným odberateľom, každý záznam sa prepošle celoplošne *(broadcast)* na všetky uzly.

- *Stream API* - aplikácia je v tomto prípade iba spracovateľská vrstva, kedy sa do Kafky napojí a vyžiada si záznamy z jednej či viacero kategórií. V aplikácií sa informácie spracujú a zašlú sa spať do Kafky do jednej či viacero kategórií.
- *Konektory (Connectors)* - jedná sa napríklad o napojenia do relačných databáz kde sa môžu zaznamenávať všetky zmeny v tabuľkách



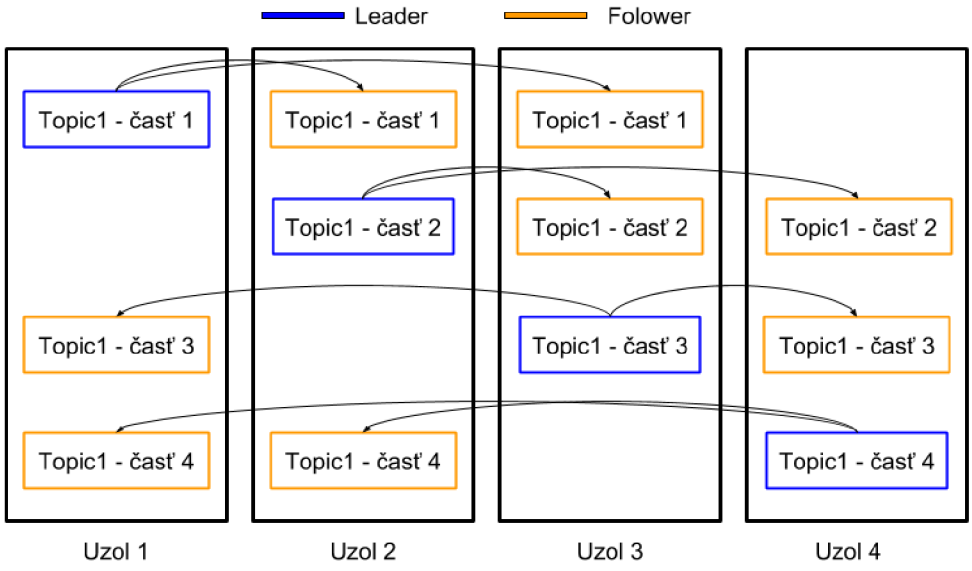
Obr. 3.1 Možnosti napojení Apache Kafka

Pre každú kategóriu Kafka udržuje tzv. logy partícií *(partitioned log)*, kde každá partícia predstavuje zoradenú a nemennú sekvenciu záznamov. Pri nahrávaní dát na konkrétnu kategóriu (topic) sa súbor rozdelený na viaceré partície môže rozprestrieť na všetky zariadenia v rámci clustera a nebyť závislý od podmienky či sa konkrétnom zariadení nachádza dostatok úložného priestoru na uloženie všetkých záznamov z konkrétnej kategórie. Každému záznamu je mimo iné pridané aj identifikačné číslo, ktoré slúži ako unikátny časový identifikátor v rámci každej partície. V dokumentácií označovanej ako *offset*. Štandardne sa *offset* posúva v čase lineárne spolu s tým, ako užívateľ načíta dáta. Toto nastavenie je ale možné manuálne pozmeniť ľubovoľne podľa užívateľa. Posunutím identifikačného čísla je teda možné dodatočné načítanie dát za celé obdobie alebo spustenie zachytávania zmien od „teraz“ [15].

Kafka uchováva uložené všetky záznamy bez ohľadu nato či boli použité alebo nie. S prihliadnutím na fakt, že úložisko má svoje limity, Kafka ponúka nastaviť dobu uchovania *(retention period)* pre uvoľnenie miesta po uplynutí nastavenej doby. Po uplynutí nastavenej doby sa záznamy automaticky vymažú [15].

Apache Kafka je technológia nasadzovaná aj vďaka jeho odolnosti voči chybám. Táto vlastnosť je dosiahnutá zálohovaním každej partície naprieč daným clusterom. Užívateľ si môže jednoducho nastaviť hodnotu replikačného faktoru. Teda číslo, ktoré uvádza na koľkých serveroch bude táto partícia zálohovaná pre prípad poruchy. Každá z partícií má server označovaný ako *leader* a k nemu môže existovať viacero odberateľov *(followers)*. *Leader* zabezpečuje všetky požiadavky na zápis i čítanie zatiaľ čo *followers* pasívne opakujú po ňom

zmenu. V prípade, že sa vyskytne chyba na serveri, na ktorom leží *leader* partície, automaticky sa vyberie jeden z *followers*, ktorý preberie funkciu primárneho člena. Každý zo serverov v rámci clustera vystupuje ako *leader* pre určitú partíciu a zároveň ako *follower* pre niektoré zo zvyšných partícií. Týmto spôsobom je zaistené rovnomerné rozprestretie záťaže naprieč clusterom [15]. Táto metóda je bližšia znázornená na obrázku 3.2. Tu je znázornený príklad kedy sa cluster skladá zo 4 serverov kde každý z nich obsluhuje určitú partíciu a replikačný faktor je v tomto prípade nastavený na hodnotu 3.



Obr. 3.2 Apache Kafka replikačný proces

Apache Kafka ponúka aj formu garancie v nasledujúcich oblastiach [15]:

- Ak sú jedným producentom poslané dva odlišné záznamy, ktoré vďaka oneskoreniu na sieti prídu v opačnom poradí, tak záznam so starším dátumom vytvorenia sa vo výpise objaví skôr
- Odberateľ vidí záznamy v poradí v akom boli do systému nahrané
- V prípade nastavenia replikačného faktora na hodnotu N, Kafka garantuje dostupnosť dát bez straty akejkoľvek z partícií v prípade, že počet serverov s poruchou neprekročí hodnotu N-1

Apache Kafka ako Systém na správu správ (Messaging System)

Kafka ponúka v tomto režime dva modely nasadenia: *queing* alebo *publish-subscribe*. Ak sa v modely *queing* dopytuje skupina odberateľov na záznam, tak server odpovie jedinou kópiou na jedného z odberateľov. Zatiaľ čo v modely *publish-subscribe* je záznam poslaný na všetkých odberateľov, ktorý si kópiu vyžiadajú [15].

Model *queing* umožňuje rozdeliť procesiacu záťaž na viacero odberateľov, keďže každý z nich disponuje iba časťou záznamov. No tento model ale nepodporuje prístup viacero odberateľov. Čo znamená, že akonáhle jeden proces načíta dáta, tie už viac nebudú dostupné.

Nevýhoda pri *publish-subscribe* vychádza z faktu, že umožňuje záznamy posielať celoplošne na viaceré zariadenia a tým pádom nedokážeme škálovať procesenie [15].

Kafka má v porovnaní s inými komunikačný systém výhodu v tom, že dokáže zaručiť doručenie záznamov v správnom poradí na viacero odberateľov. Je to dosiahnuté priradením konkrétnej partície v kategórií *(topic)* na jedného z odberateľov z každej skupiny. Čo má za následok, že každá partícia je odoberaná práve jedným konzumentom [15].

Apache Kafka ako úložisko (Storage System)

Dáta, ktoré sú nahrané na disk v systéme Apache Kafka sú replikované na viaceré zariadenia v závislosti na replikačnom faktore. Producenti, ktorí nahrávajú záznamy do Kafky majú možnosť byť informovaný prostredníctvom *acknowledge* správ z Kafky, že zápis prebehol v poriadku a je plne zreplikovaný na vybrané servre [15].

Spôsob ukladanie dát v Apache Kafka funguje rovnako dobre či sa ukladajú záznamy o veľkosti 80KB alebo 8TB [15].

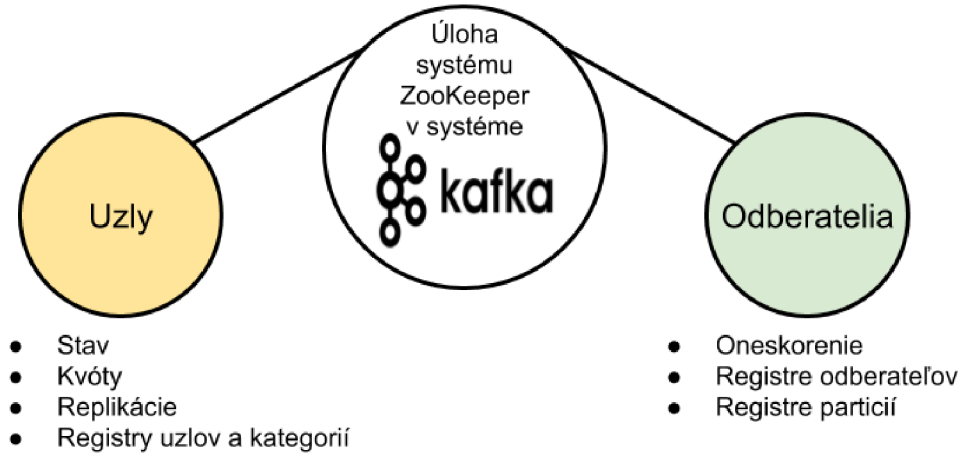
Apache Kafka ako Spracovanie toku dát (Stream Processing)

Pod spracovaním toku dát priamo v systéme Kafka sa rozumie proces, kedy sa na vstupe príjmu dáta, vykonajú sa nad ním určité operácie a takto poopravený tok dát sa pošle na výstup a to všetko v reálnom čase [15].

Jednoduché procesenie dovoľuje aj nasadenie API (Application Programming Interface) volaní na rozhranie producentov a odberateľov. V prípade komplexnejších úprav Kafka ponúka *Streams* API, ktoré umožňuje počítať agregácie nad tokom dát alebo spájať toky dát do jedného celku. Pomáha taktiež pri problémoch, ktoré aplikácie na spracovanie toku dát majú ako napríklad reorganizáciu v prípade zlého poradia dátových záznamov a vykonávanie stavových operácií [15].

3.2 Apache Zookeeper

Apache Zookeeper ako distribuované, open-source koordinačné riešenie je nasadzované ako správca clustera a je neoddeliteľným pomocníkom pri práci s Apache Kafka. V dokumentácií pri spustení Kafka servera sa inštancia Apache Zookeepera udávaná ako podmienka pred spustením [15]. Na obrázku číslo 3.3 je znázornená úloha Apache Zookeeper pri nasadení spolu s Kafkou.



Obr. 3.3 Apache Zookeeper a jeho úlohy

Kafka využíva služieb Apache Zookeeper na koordináciu uzlov *(brokers)* a celého clustera. Ako je znázornené na obrázku 8, Apache Zookeeper pomáha Kafke v 2 oblastiach- pri správe uzlov *(brokers)* a odberateľov *(consumers).* V prípade uzlov sa jedná o nasledujúce oblasti [16]:

- *Stav (State)* - podáva uzlom informácie o tom či niektorý z uzlov zlyhal. Kontrola prebieha vďaka neustálemu posielaniu tzv. *heartbeats* žiadostí. Ak nastane situácia, kedy sa jeden uzlov odpojí, Zookeeper má na starosť výber nového vodcu skupiny odberateľov *(leader)*.
- *Kvóty (Quotas)* - dovoľuje niektorým uzlom mať odlišné kvóty na produkovanie a spracovanie záznamov.
- *Replikácie (Replicas)* - Zookeeper uchováva tzv. ISR *(in-sync replicas)* replikácie pre každú z kategórií. Z názvu vyplýva, že sa jedná o synchronizovanú replikáciu naprieč clusterom. To znamená, že ak jeden z uzlov zlyhá, ISR záznam je aktualizovaný. Ak uzol, ktorý zlyhal bol vodcom (leader) v jednej z kategórií, Zookeeper je zodpovedný za výber nového vodcu.
- *Registry uzlov a kategorií (Nodes and Topics Registry)* - Zookeeper uchováva registre uzlov a kategórií. Vďaka týmto informáciám sa vieme dostať k údajom ako zoznam všetkých dostupných uzlov v rámci clustera spolu s listom kategórií, ktoré drží konkrétny uzol. Tieto záznamy ale nie sú trvalé- udržiavajú sa aktívne po dobu spojenia so systémom Zookeeper. Sú zahodené akonáhle sa spojenie ukončí

Druhá oblasť využitia spolu s Apache Kafka je v oblasti správy odberateľov a to v nasledujúcich bodoch [16]:

- *Oneskorenie (offset)* - Zookeeper je predvolené úložisko pre nastavenia oneskorenia pre konkrétneho odberateľa. V budúcnosti sa plánuje na tento účel dedikovaný Kafka uzol nazvaný *Offset Manager*
- *Registre odberateľov (Registry)* - odberatelia majú taktiež, podobne ako uzly, vlastný register a aplikujú na ne rovnaké pravidlá. To znamená, že je záznam o ňom vymazaný akonáhle sa odberateľ odpojí.
- *Registre partícií (Partitions registry)* - keďže je partícia viazaná vždy na práve jedného odberateľa, ktorý spadá do konkrétnej skupiny, systém potrebuje vedieť väzbu medzi partíciou a odberateľom. Táto informácia sa uchováva v tomto registri a podobne ako predošlé registre, aj tento sa vymaže v momente, keď partícia prestane existovať.

3.3 Apache Cassandra

Apache Cassandra bola vytvorená spoločnosťou Facebook s prvotnou myšlienkou na rýchle prehľadávanie prichádzajúcej pošty. V roku 2008 bola Cassandra predstavená ako open-source riešenie. Od tej doby sa ako celosvetovo nasadzovaná NoSQL databáza sa presadila najmä vďaka jeho vlastnostiam medzi ktoré patria [17] :

- *Ľahká škálovateľnosť (Elastic scalability)* - v prípade potreby je možné jednoducho pridať ľubovoľné množstvo hardvéru na uchovanie potrebného množstva dát
- *Vysoká dostupnosť (Always on architecture)* - ak dôjde k výpadku jedného z uzlov, systém zostane funkčný. V dokumentácií je tento stav označovaný aj ako *single point of failure*.
- *Vysoký lineárna škálovateľnosť výkonu (Fast linear-scale performance)* - pridávaním uzlov do clustera sa lineárne zrýchli odozva systému
- *Flexibilné úložisko (Flexible data storage)* - Cassandra dokáže uchovať široké množstvo dát zahrnujúce dáta typu- štruktúrované, semi-štruktúrované či neštruktúrované. Dynamicky sa prispôsobí potrebám použitia.
- *Jednoduchá distribúcia dát (Easy data distribution)* - v prípade potreby sa jednoducho dajú replikovať dáta naprieč clusterom
- *Rýchly zápis (Fast writes)* - Cassandra bola od počiatku dimenzovaná na beh nad lacnejším hardvérom. Preto ponúka rýchly zápis a čítanie terabajtov dát bez veľkého vplyvu na výkon

NoSQL databázi existuje viacero typov a Apache Cassandra využíva typ ukladania tzv. *Wide Column Store*. Tento typ databázy používa, podobne ako relačné databáze, riadky a stĺpce. Rozdiel spočíva v tom, že formát a názov stĺpca sa môže líšiť riadok od riadku v rámci jednej tabuľky [18].

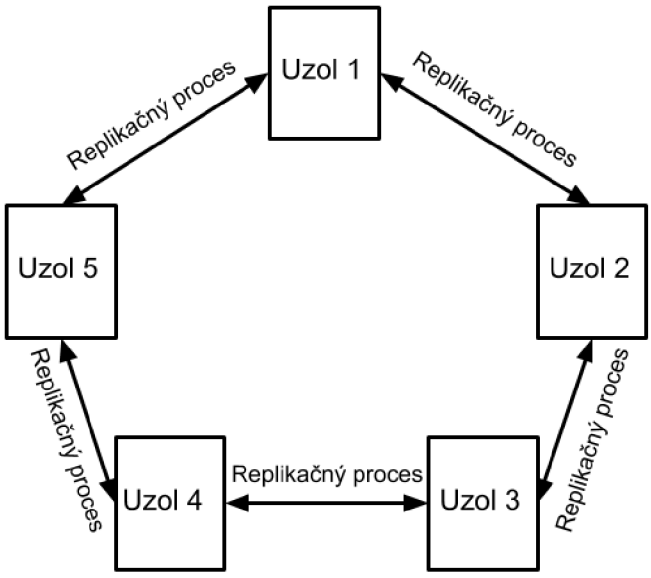
Vo všeobecnosti, NoSQL databáze poskytujú možnosť ukladania dát odlišne od spôsobu, ktorý poznáme v klasických relačných databázach. Takto uložené dáta nemajú

schému, poskytujú jednoduchú implementáciu API rozhraní, nepodporujú transakcie a zvládnu pojmúť veľké množstvo dát. Pod pojmom „transakcia“ sa rozumie časť práce vykonaná nad databázou, ktorá môže byť v prípade chyby alebo správne nedokončenej operácie vrátená späť do pôvodného stavu [17].

Cassandra sa používa sa v distribuovaných systémoch na spracovanie veľkého množstva dát naprieč serverami umiestnených v decentralizovanom clustery. Architektúra databázy Cassandra stavia na uzloch, ktoré charakterizujú nasledujúce prvky [17]:

- každý uzol v clustery figuruje ako nezávislá jednotka, no zároveň je prepojená so všetkými ostatnými uzlami v clustery
- každý uzol môže prijať požiadavku na zápis či čítanie z úložiska napriek tomu, že sa daný záznam sa nemusí nenachádzať na tomto prvku
- ak nastane výpadok uzlu, požiadavky na zápis či čítanie môže odbaviť niektorý z iných uzlov v clustery

Podobne ako Apache Kafka, aj Cassandra replikuje konkrétny záznam naprieč rôznymi uzlami v clustery. Ak nastane situácia, kedy jeden z uzlov má neaktuálny záznam, Cassandra zabezpečený aktualizáciu záznamu. Na komunikáciu medzi uzlami alebo zistenie či niektorý z uzlov v clustery zlyhal sa používa na pozadí protokol *Gossip*. Na obrázku 3.4 je znázornená metóda replikácie [17]:



Obr. 3.4 Replikačná schéma Apache Cassandra

Medzi kľúčové komponenty Apache Cassandra sa zaraďujú nasledujúce časti [17]:

- *Záznamy priebehu (Commit log)* - je to mechanizmus, ktorý sa využíva pri obnovený po zlyhaní systému. Taktiež sa sem ukladajú záznamy o každom zápise v systéme
- *Mem-tabuľka (Mem-table)* - po uložení záznamov v *Commit Log* sa taktiež ukladajú do Mem-tabuľky, ktorá je typu *memory-residant*. To znamená, že zostáva v pamäti po celú dobu behu systému.

- *SSTabuľka (SSTable)* - tabuľka, kde sa nahrávajú dáta z Mem-tabuľky v prípade, že zaplnenie Mem-tabuľky dosiahne určitú hodnotu.
- *Bloom filter* - nedeterministický algoritmus na testovanie či element je súčasťou setu. Je dostupný po každom volaní dopytu *(query).*

Užívatelia sa komunikujú s Apache Cassandra pomocou *Cassandra Query Language (CQL)*. Pri práci s databázou sa používa pojem *Keyspace*, ktorý predstavuje istý kontajner tabuliek. Programátori používajú nadstavbu- jazyk *CQLSH.* Ide o rozšírenie postavené nad jazykom Python určené k spustenie CQL príkazov [17].

Pri vytváraní kontajneru *(keyspace)* sa definujú 2 základné parametre: replikácie *(replication)* s definovaným replikačným faktorom a trvalosť zápisu *(durable writes)*. Pri definovaní replikačnosti poznáme 2 stratégie:

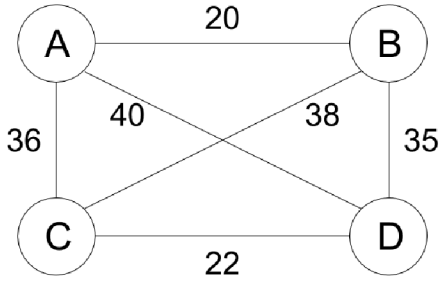
- *Jednoduchá stratégia (Simple Strategy)* - špecifikuje hodnotu replikačného faktoru pre celý cluster
- *Stratégia sieťovej topológie* - možnosť nastavenia rozdielneho replikačného faktoru pre každé dáta centrum oddelene

4 PROBLEMATIKA RIEŠENÝCH LABORATÓRNYCH ÚLOH

V tejto kapitole je priblížená problematika obchodného cestujúceho a jeho zjednodušené riešenie, ktoré je neskôr využité aj v laboratórnej úlohe. Cieľom úlohy je demonštrovať zrýchlenie výpočtu v clustery. Algoritmus, z ktorého sa v tejto práci vychádzalo pochádza z voľne dostupného repozitáru pod nasledujúcim odkazom [24]. Zvyšné dve laboratórne úlohy, ktoré nesúvisia s Problémom obchodného cestujúceho tu nie sú rozpísané z dôvodu toho, že technológie v nich použité sú rozpracované v samostatných kapitolách a princíp rozdelenia výpočtu medzi uzly zostáva rovnaký.

4.1 Problém obchodného cestujúceho

Jedná sa problém, ktorý sa zjednodušene popisuje nasledovne: „Ak dostanem konečný zoznam miest a vzdialenosť medzi každou dvojicou z nich, ktorá cesta bude najkratšia s tou podmienkou, že chcem navštíviť každé mesto práve jeden krát a vrátiť sa späť do miesta, z ktoré som moju cestu započal?“. Tento problém je považovaný za jeden z najznámejších optimalizačných úloh a zaraďuje sa do kategórie NP-úplný. Čo značí, že s rastúcim počtom miest, cez ktoré má trasa prechádzať rastie náročnosť riešenia exponenciálne [22]. Mnoho krát sa pri vysvetľovaní znázorňuje tento problém pomocou obojsmerného grafu, kde uzly predstavujú miesta a hrany uvádzajú cestu. Každá z hrán má pridelené číslo, ktoré reprezentuje napr. vzdialenosť alebo náklady na cestu. Jeden z príkladov môže byť graf na nasledujúcom obrázku



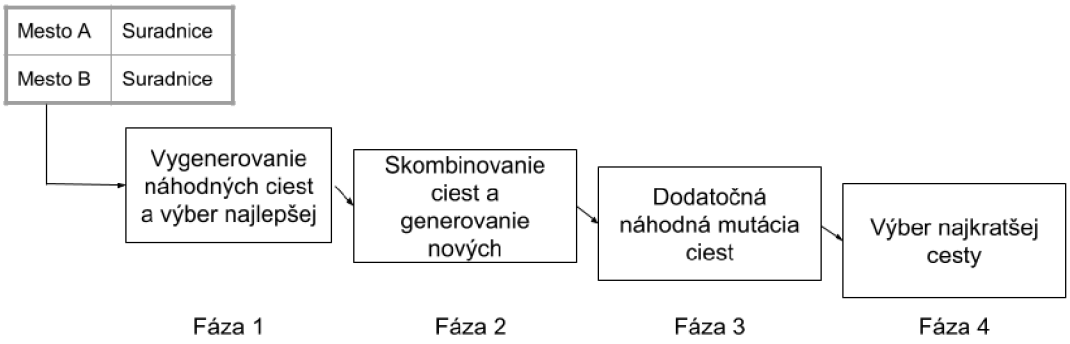
Obr. 4.1 Ukážka grafu so 4 uzlami a rôzne ohodnotenými hranami

V odbornom článku [23] sa spomína ako jedna z možností na riešenie tohto problému rozdelenie miest (teda uzlov v ponímaní grafových schémat) do menších celkov, ktoré sa ku koncu spoja do konečného riešenia. PNP *(Pick Near Point)* algoritmus potom pomôže pri spájaní týchto menších celkov dohromady. Výhodou tohto systému v porovnaní s ostatnými je, že výsledok je závislí na porovnaní so všetkými uzlami.

Riešenia v tejto práci je optimalizované na základe genetického algoritmu. Čím viac času umožníme programu hľadať najlepšiu cestu, tým presnejšie sa k reálne najkratšej ceste dostaneme. To ale neznamená, že výsledok, ktorý nám program poskytne je najoptimálnejšia trasa. Preto sa pri spustení definuje maximálny čas na výpočet. Po jeho uplynutí dostanete výsledok, ktorý sa považujete za dostatočne dobrý. Algoritmus generuje náhodne cesty medzi

miestami, nájde najkratšiu a kombinuje ich s ostatnými cestami za účelom dosiahnutia ešte výhodnejšej trasy.

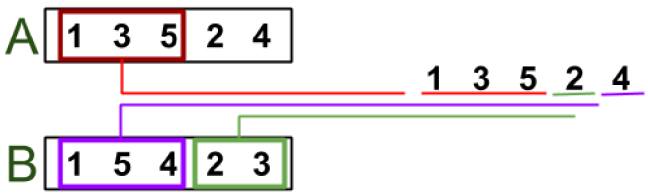
Bližšie vieme popísať algoritmus pomocou obrázka číslo 4.2.



Obr. 4.2 Algoritmus TSP v zjednodušenom diagrame

Pri štarte programu si užívateľ definuje počet miest cez ktoré má trasa prechádzať a taktiež tu existuje možnosť náhodného výberu miest programom. Prvé zadané, resp. vybrané miesto sa považuje za východzie aj návratové miesto. V prvej fáze programu sa vygeneruje základný zoznam možných ciest. V programe sa na tento zoznam používa označenie *Generation Zero*. Na vstup funkcie, ktorá tento zoznam generuje, sa privedie zoznam miest, cez ktoré má trasa prechádzať a funkcia vytvorí 100 náhodných možných prechodov. Prechod s najkratšou dĺžkou sa považuje za základnú hodnotu, s ktorou sú neskoršie nájdené cesty porovnávané.

Fáza dva generuje nové cesty pomocou jednoduchého prenásobenia. V terminológií genetického algoritmu sa jedná o časť nazývanú „kombinácie“. V tomto algoritme sa každá trasa A skombinuje s každou inou alternatívnou cestou B a to tak, že sa prvá polovica trasy A zostáva nezmenená a pridá sa k nej druhá polovica trasy B. Ak sa v druhej polovici trasy B nachádza už miesto, ktorým už trasa A prechádza, zoberú sa zvyšné unikátne miesta z trasy B a novo vzniknutá cesta sa doplní miestami z prvej polovice trasy B. Tento proces je znázornený aj na obrázku číslo 4.3 nižšie:



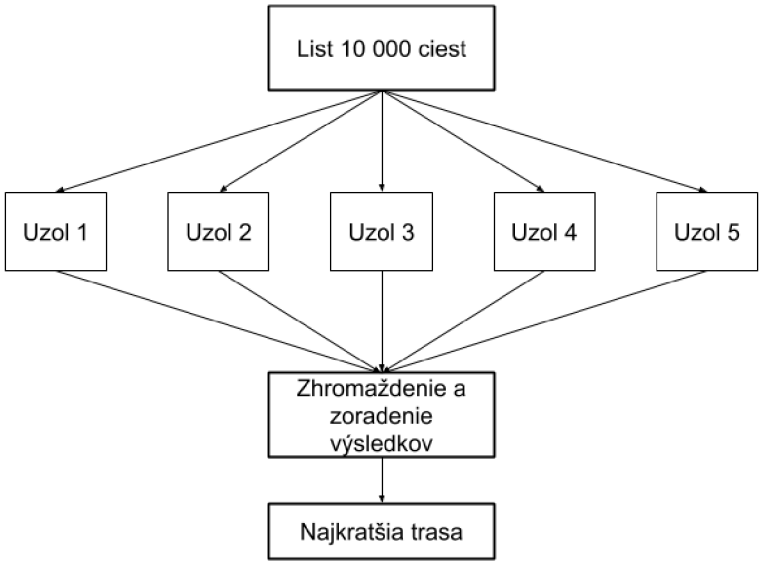
Obr. 4.3 Algoritmus na kombinovanie ciest

Takto novo vzniknutá cesta sa ešte dodatočne upraví vo fáze tri. Z pohľadu názvoslovia genetického algoritmu sa jedná o fázu „mutácie“ generácie. Vygenerujú sa dve náhodné pozície a čísla na týchto pozíciách sa prehodia. Týmto náhodným premiešaním je zabezpečené náhodnejšie generovanie nových ciest a dosiahnutie ďalších kombinácií. Keďže pre väčšie

skupiny miest nemusí byť dostatočné generovanie ciest vo fázy jedna.

V poslednej fázy sa zoznam ciest zoradí a vyberie sa najkratšia. Postup sa opakuje, pokiaľ sa neprekročil maximálny stanovený čas na výpočet alebo sa nedosiahol maximálny počet uskutočnených iterácií.

Samotná paralelizácia výpočtu spočíva v tom, že vytvorený zoznam náhodných kombinácií prechodov medzi mestami je rozdelený medzi uzly. To, koľko práce bude daný uzol vypočítavať závisí podľa výkonnostných možností každého uzlu. Ako príklad uvediem, že máme cluster s piatimi rovnocennými výpočtovými uzlami a chceme spočítať prejdenú vzdialenosť pre náhodne vygenerovaných 10 000 ciest.



Obr. 4.5 Algoritmus pri paralelizácií

Ako je patrné z obrázka 4.5, v tomto prípade dostane každý z uzlov prepočítať vzdialenosť pre približne 2000 ciest, keďže sa list 10 000 ciest rozdelí medzi päť výpočtových uzlov. Výstupom každého z uzlov bude objekt typu *List<Route>*, ktorý prichádza na vstup funkcie pre zhromažďovanie výsledkov. Tu sa výsledky spoja do jedného objektu, ktorý sa v zapätí zoradí podľa vypočítanej dĺžky. Prvá cesta z tohto zoradeného zoznamu sa porovná s doposiaľ najkratšou nájdenou. V prípade, že je prejdená vzdialenosť menšia, nová cesta sa nastaví ako najlepšia. Fitness funkcia, ktorá porovnáva výkonnosť každej dosiahnutej generácie, je v tomto prípade teda funkcia určujúca prejdenú vzdialenosť trasou. Program následne pokračuje v ďalšom hľadaní pokiaľ sú splnené požiadavky na počet iterácií alebo celkový vypočítavaný čas.

5 LABORATÓRNE ÚLOHY

Obsahom tejto kapitoly je ukážka konkrétnych zadaní laboratórnych úloh v oblasti spracovania toku dát v reálnom čase pre predmet „*Paralelní zpracování dat*“, ktorý bude vedený v českom jazyku. Preto aj úlohy uvedené nižšie sú písané v českom jazyku. Informácie, ktoré sú uvedené v popise laboratórnych úloh sú čerpané z citácií uvedených už vo vyšších častiach tejto práce. Z tohto dôvodu neuvádzam v zátvorkách odkaz na citácie.

5.1 Úvod do technologií Apache Kafka a Apache Cassandra

Cíl úlohy

Cílem této laboratorní úlohy je přiblížit přínos technologií Apache Kafka a Apache Cassandra při práci s velkými daty. Student bude mít možnost si vyzkoušet prvotní nastavení těchto databázi a provést s nimi základní operace.

Zadání

- Seznámení s technologiemi, možnostmi nasazení a jejich přínos při zpracovávaní toku dat v reálném čase
- Vyzkoušejte jednoduchý program na generování a zachytávaní zpráv v prostředí Apache Kafka. Vysvětlení základních potřeb Apache Zookeper pro funkci Kafky
- Práce se základními příkazy v prostředí Apache Cassandra

Teoretický úvod

Z výhod Apache Kafka těží společnosti jako LinkedIn, Yahoo, Twitter, Spotify, Airbnb či Uber například na zachytávaní logovacích zpráv [13]. Apache Cassandra jako NoSQL databáze je nasazená ve firmách jako GitHub, NetFlix či eBay na ukládání velkých datových sad [14]. Obě technologie mají společnou vlastnost, že se v praxi nasazují ve výpočtových clusterech z důvodu rozdělení zátěže a rychlejší dostupnosti.

V případě Apache Kafka se nejedná primárně o databázový systém na ukládání dat, ale jako nástroj na zpracování přicházejících toků dat. V otázce produkčního nasazení známe dva typy použití [15]:

- mezivrstva fungující v reálnem čase, která spolehlivě doručí data mezi systémem, který zprávy generuje a aplikací, která je zpracuje
- jako aplikace, která v reálnem čase transformuje nebo reaguje na přicházející tok dat na vstupu

Na jedné instanci Kafky se může odchytávat několik procházejících zdrojů, které jsou od sebe navzájem odlišné kategorie *(topics)*. Tímto způsobem se informace z několika aplikací působících na rozdílných zařízeních mohou shromažďovat v jedné kategorii. Na práce s kategoriemi Apache Kafka podporuje 4 různé typy API připojení [15]:

- *Producenti (Producer API)* - táto API umožní aplikacím nahrávat data do jedné nebo více kategorií *(topics)*
- *Odběratelé (Consumer API)* - aplikace mají možnost se přihlásit k odběru informací na jedné nebo více kategorií. Každý odběratel se zařadí do určité skupiny odběratelů a každý záznam přidaný do kategorie se pošle jenom jednou na každou z odběratelských skupin.
- *Stream API* - aplikace je v tomto případě jenom zpracovatelská vrstva, kdy se do Kafky napojí a vyžádá si záznamy z jedné nebo více kategorií. V aplikaci se informace zpracují a zašlou se zpátky do Kafky do jedné nebo více kategorií
- *Konektory (Connectors)* - jedná se například o napojení do relačních databází, kde se mohou zaznamenávat všechny změny v tabulkách

Vysoká dostupnost je dosáhnuta zálohováním každého oddílu napříč daným clusterem na základě hodnoty replikačního faktoru (číslo uvádějící počet záloh daného oddílu pro případ poruchy). Každý z oddílu má server označovaný jako *leader* a k němu může existovat více odběratelů *(followers)*. *Leader* zabezpečuje všechny požadavky na zápis i čtení, zatímco *followers* pasivně opakují změnu po něm. Každý ze serveru v rámci clusteru vystupuje jako *leader* pro určitý oddíl a zároveň jako *follower* pro některé ze zbývajících oddílu [15].

Pro běh Apache Kafka je potřebné spuštění Apache Zookeeper coby distribuované a koordinační řešení na zprávu clusteru a koordinaci uzlů *(brokers)*. Mezi některé z jeho úkolů patří podávání informace uzlům o dostupnosti/ selhání jiného uzlu, výběr nového *leadera* pro konkrétní kategorii v případě výpadku toho původního uzlu, nebo uchovává záznamy o tom, které kategorie má daný server u sebe uložené [16].

V roce 2008 byla Apache Cassandra představena jako open-source databázové řešení. Od té doby se jako celosvětově nasazovaná NoSQL databáze prosadila zejména díky jeho vlastnostem jako lehká škálovatelnost, vysoká dostupnost (absence *single point of failure*) a rychlý zápis, jelikož od počátku byla dimenzovaná na fungování i na levnějším hardwaru [17].

Mezi klíčové komponenty Apache Cassandra patří [17]:

- *Záznamy průběhu (Commit log)* - je to mechanizmus, který se využívá při selhání systému pro obnovení. Taktéž se sem ukládají záznamy o každém zápisu v systému
- *Mem-tabulka (Mem-table)* - po uložení záznamu v Commit Log se ukládají do Mem-tabulky, která je typu *memory-residant*. To znamená, že zůstává v paměti po celou dobu běhu systému.
- *SSTabulka (SSTable)* - tabulka, kde se nahrávají data z Mem-tabulky v případě, že zaplnění Mem-tabulky dosáhne určité hodnoty.
- *Bloom filter* - nedeterministický algoritmus na testování, jestli je element součástí sady. Je dostupný po každém volaní dotazu *(query)*.

NoSQL databází existuje několik typů a Cassandra využívá typ ukládání tzv. *Wide Column Store*. Tento typ databází používá, podobně jako relační databáze, řádky a sloupce. Rozdíl spočívá v tom, že formát a název sloupce se může lišit řádek od řádku v rámci jedné tabulky [18].

Ve všeobecnosti, NoSQL databáze poskytuje možnost ukládání dát odlišně od způsobu, který známe z klasických relačních databázích. Takto uložené data nemají schéma, poskytují jednoduchou implantaci API *(Application Programming Interface*) rozhraní, nepodporují transakce a zvládnou pojmout velké množství dat. Pod pojmem „transakce“ se rozumí část práce vykonaná v databázi, která může být v případě chyby nebo nesprávně dokončené operace, vrácena zpátky do původního stavu [17].

Cassandra se používá v distribuovaných systémech na zpracování velkého množství dat napříč servery umístněnými v decentralizovaném clusteru. Jeho uzly figurují jako nezávislé jednotky, ale zároveň jsou propojené se všemi zbývajícími uzly v clusteru. Každý z nich může přijmout požadavek na zápis nebo čtení z úložiště navzdory tomu, že se daný záznam nemusí nacházet na tomto prvku. V případě výpadku uzlu, požadavky na zápis nebo čtení může odbavit některý z jiných uzlu v clusteru [17].

Cassandra replikuje konkrétní záznam napříč různými uzly v clusteru. Pokud nastane situace, kdy jeden z uzlů nemá aktuální záznam, Cassandra zabezpečí aktualizaci záznamu. Na komunikaci s Apache Cassandra slouží *Cassandra Query Language (CQL)*. Při práci s databázi se používá pojem *Keyspace*, který představuje nějaký kontejner tabulek. Programátoři používají nadstavbu- jazyk *CQLSH.* Jde o rozšíření jazyku Python, který je určen k spuštění CQL příkazu [17].

Všechny skripty, které dnes budete používat jsou typu .bat na běh v prostředí DOS anebo Windows. Známé jsou spíše pod označením *batch* soubory. Jedná se o sérii příkazů, které můžete najít v příkazovém řádku *(Command Line)*. Příkazy se jednoduše napíší v textovém editoru a uloží se s příponou .bat.

Pracovní postup

Z E-learningu si stáhnete soubor *Kafka* a uložíte si ho ve své složce na počítači. Jak bylo v úvodu zmíněnu, spuštění Apache Zookeeper je podmínka pro běh Apache Kafka.

1. Konfigurace a spuštění Apache Zookeeper

Ve složce *\config* najděte soubor *zookeeper.properties*., otevřete ho a zkontrolujte, že je klientský port nastavený na hodnotu 2181. Spusťte si příkazový řádek a přejděte pomocí příkazů „*cd*“ do podsložky *\bin\windows* (v rámci složky Kafka), kde jsou alokované skripty pro práci s Kafkou. Zadejte následující příkaz:

„zookeeper-server-start.bat ..\..\config\zookeeper.properties“

Vytvoří se instance Apache Zookeeper spolu s vlastnostmi definovanými v souboru *zookeeper.properties*.

2. Konfigurace a spuštění Apache Kafka

Postupujte podobně jako při prvním bodu a upravte soubor *server.properties.* V tomto konfiguračním souboru Kafky se nachází unikátní identifikátor pro každý uzel v clustery, cesta k logovacím záznamům, IP adresa a port, kde leží instance Apache Zookeeper a mnohé další

nastavení. Ujistěte se proto, že sekce dokumentu s názvem *Zookeeper* obsahuje řádek *zookeeper.connect=localhost:2181*. V novém okně příkazového řádků přejděte opět do podsložky *\bin\windows* a zadejte příkaz:

„kafka-server-start.bat ..\..\config\server.properties“

3. Vytvořte novou kategorií (topic)

Otevřete si nové (třetí) okno příkazového řádku a opět přejděte do podsložky *\bin\windows* a pomocí následujícího příkazů vytvořte novou kategorii:

„kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic XY“

Tento příkaz vytvoří kategorii s vaším názvem, která bude mít jediný oddíl a nebude se replikovat na žádný jiný uzel v rámci clusteru. V příkazu nahraďte „XY“ svým názvem kategorie. Vytvoření kategorie si můžete ověřit pomoci příkazů:

„kafka-topics.bat --list --zookeeper localhost:2181“

4. Spuštění odběratele správ (consumer)

V posledním otevřeném okně zadejte text:

„kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic XY --from-beginning“

Toto způsobí vytvoření odběratele, který bude naslouchat novým záznamům na lokálním zařízení na portě 9092 a bude je ukládat do kategorie, kterou jste si vytvořili. Příznak *from beginning* hovoří o tom, že bere do úvahy všechny záznamy, které na danou kategorii přišly, a nejen ty od spuštění skriptu.

5. Generovaní správ na základě změn v souboru

Pro příklad reálného nasazení, budeme uvažovat případ, že program bude kontrolovat a zapisovat nové záznamy v souboru, který může představovat logovací záznamy jakékoli aplikace. K tomuto účelu je potřebné pozměnit dva konfigurační soubory. Přejděte proto opět do podsložky *\config* a otevřete soubor *connect-file-source.properties*. Konfigurační položka *file* určuje název souboru (napište například source.txt), který se bude skenovat a položka *topic* název kategorie, na kterou se nové záznamy ze souboru budou posílat. Sem zadejte název své kategorie.

Druhý soubor *connect-file-sink.properties* upravte následovně - do položky *topics* napište název kategorie, jejíž zprávy se budou ukládat, a do řádku *file* napište název souboru, do kterého se nové záznamy budou logovat. V podsložce *\bin\windows* vytvořte textový soubor s názvem, který jste definovali v *connect-file-source.properties.*

Otevřete si další okno příkazového řádku v podsložce *\bin\windows* a spusťte následující příkaz:

„connect-standalone.bat ..\..\config\connect-standalone.properties ..\..\config \connect- file-source.properties ..\..\config\connect-file-sink.properties“

Ověřte, že při jakékoli změně zdrojového souboru se nový záznam objeví ve výpise odběratele zpráv.

6. Spuštění a základní konfigurace Apache Cassandra

Skripty na spuštění databází najdete v složce *Cassandra* na disku *D:*. Spusťte skript *Start_Cassandra*. Po úspěšném spuštění si na pracovní ploše spusťte aplikaci *Cassandra CQL Shell (CQLSH)*. Mezi základní nastavení patří vytvoření *keyspace*, tedy jistého kontejneru tabulek. Vytvořte nový kontejner následujícím příkazem:

„CREATE KEYSPACE XY WITH replication = {'class':'SimpleStrategy', 'replication_factor' : 1};“

Znovu nahraďte *„XY“* vámi zvoleným názvem a ponechejte nastavení replikačního faktoru na hodnotě 1.

7. Vytvoření tabulky a základní SQL operace v Apache Cassandra

Operace nad tabulkami se do velké míry podobají jazyku SQL. Proto je vytvoření nové tabulky dost intuitivní, jelikož tabulka musí existovat v rámci určitého kontejneru, napište následující příkaz do rozhraní *Cassandra CQL Shell: „USE XY“.* Opět nahraďte *„XY“* vaším názvem kontejneru. Pro vytvoření ukázkové tabulky použitě text:

„CREATE TABLE XYZ(XYZ_id int PRIMARY KEY, XYZ_name text);“

Opět zaměňte výraz „XYZ“ vlastním názvem tabulky. Klíčové slovo *PRIMARY KEY* specifikuje, že v daném sloupci se nacházejí unikátní hodnoty. Tento stav je ideální pro tvorbu indexů. Teda formu vnitřního uspořádání, která umožní rychlejší průběh tabulkou při hledání konkrétních hodnot.

Nahrajte do tabulky několik záznamů na ukázku. Zkopírujte a pusťte proto tyto příkazy s vámi vybranými názvy:

„INSERT INTO XYZ (XYZ_id, XYZ_name) VALUES (1, ‘test_1‘);
INSERT INTO XYZ (XYZ_id, XYZ_name) VALUES (2, ‘test_2‘);“

Výsledek těchto příkazů zkontrolujte pomocí:

*„SELECT * FROM XYZ“*

Zkuste dodatečně přidat záznam s parametry, například *(2, ‘test_3‘)*. Vrátí vám systém chybu odkazující na už existující primární klíč? Jestli ne, co se změnilo v tabulce?

5.2 Zpracovaní dát z Twitteru v reálnem čase

Cíl úlohy

Cílem této laboratorní úlohy je vytvoření a úprava kódu v jazyce Java. Kód slouží na napojení do systému společnosti Twitter a analyzovat záznamy, které generují uživatelé po celém světě. V druhé části úlohy bude demonstrovaný paralelní výpočet nad daty, které studenti nahrají do platformy Apache Kafka.

Zadání

- Vytvořte kód v jazyku Java a napojte se na vývojářské prostředí společnosti Twitter
- Analyzujte výstup v reálním čase pomocí vestavěných funkcí Apache Structured Streaming
- Výstup nahrajte do databáze Apache Cassandra
- Spusťte posílaní správ do rozhraní Apache Kafka běžícím na master uzle, nad kterými je demonstrované paralelní zpracování dat

Teoretický úvod

Jelikož úvod do problematiky Apache Kafka a Cassandra jste měli v předcházejícím cvičení, v tomto teoretickém úvodu se zaměříme na seznámení s open-source technologií Apache Structured Streaming a na základní operace s ní. Jde o technologii postavenou nad jádrem Apache Spark.

Apache Spark je výpočtový systém pracující v clusteru, kterého největší výhody jsou rychlost a široké možnosti nasazení. Nabízí API rozhraní (Application Programming Interface) v jazycích Java, Scala, Python či R. V porovnání s platformou Hadoop nabízí významnou výhodu v podobě rychlosti načítaní dat. Dokud Hadoop svoje data a průběžné výpočty ukládá na disk, Apache Spark může pracovat v tzv. „in-memory“ režimu. To znamená, že na ukládání využívá operační paměť, která je rychlejší. Mezi nástroje patří mimo jiné i nástroje jako Spark SQL na tvorbu dotazu v jazyku SQL a procezení strukturovaných dat nebo knihovna MLlib pro strojové učení [8].

Apache Structured Streaming je vcelku nová technologie a rozdíl v porovnání se starším přístupem Apache Streaming vychází už z jeho názvu – Structured, tedy strukturovaný. Při souvislém neohraničeném toku dat do systému se postupně záznamy přidávají jako řádky do tabulky [12]. V dokumentací dále označovaná jako *Input table*. Funkce na vstupu (například počítaní konkrétních slov) potom generuje z dat ve vstupní tabulce tzv. *Result Table*, která se posílá dále na výstup. Při počátečním nastavování aplikace se taktéž definuje jeden z následujících módu, v jakém bude pracovat [12]:

- *Complete mode* - celá aktualizovaná tabulka je poslaná na výstup. Závisí od napojení do cílového úložiště *(storage connector)* jak zpracuje celou tabulku
- *Append mode* - na výstup se pošlou jenom záznamy z Result Table, které tam přibyly od posledního zápisu, a neočekává se při nich změna
- *Update mode* - tento mód zachytí jenom změny *(updates)* v Result Table, které proběhly od posledního zápisu do úložiště

V reálnem nasazení je možné, že děláme analýzu v minutových intervalech a jeden ze záznamů se do systému dostane se zpožděním. To se pozná na základě časového razítka ve vstupní tabulce *(Input table)*. Structured Streaming v tomto případě jednoduše přepočítá toto agregační okno a na výstup pošle informaci o nové hodnotě. Systém však povoluje definovat časovou zarážku. Tedy čas, jak dlouho má systém ještě udržovat staré agregační hodnoty v operační paměti pro případ zpožděních záznamů. Jestliže záznam dorazí na vstup po uplynutí této zarážky, bude ignorovaný. Tato funkce je označovaná jako *watermarking* [12].

Mezi podporované formáty souboru, s kterými umí Structured Streaming pracovat, patří už zmiňovaný Apache Kafka nebo klasické soubory typu JSON, CSV, parquet a UTF-8 text ze socket připojení. Při klasických typech souborů je nutné napřed definovat jejich vnitřní schéma [12].

Při odstraňování duplikačních záznamu v datovém toku se využívá identifikátor. Při práci s nimi odlišujeme dva přístupy [12]:

- *S watermark* - jestliže máme definované časové období, za které se duplikované záznamy mohou objevit, tak umíme využít sloupec s časovým razítkem a jedinečný identifikátor na to, abychom zmenšili množství unikátních stavů, které systém musí udržovat a kontrolovat
- *Bez watermark* - jelikož neexistuje časová zarážka, za kterou systém kontroluje možné stavy, systém musí udržovat kompletní historický seznam možných stavů. Podobně jako Spark Streaming, tak i Structured Streaming nabízí možnost obnovy po chybě anebo nečekaném selhání. V Spark dokumentaci známe jako *checkpointing*. Při výstupu můžeme jednoduše nastavit cíle na fyzickém stroji, kde se informace o stavu běhu systému a probíhajících agregačních funkcích budou ukládat. Tato možnost se definuje při výstupní funkci *writeStream* zadáním volitelného parametru *checkpointLocation*.

Pracovní postup

Z E-learningu si stáhněte soubor LAB2.rar. Uložte si ji ve své složce na počítači. Jedná se o předpřipravené programy v jazyku Java. Otevřete si projekt u Vás na počítači v programu Eclipse.

1. Spuštění Apache Kafka

Na disku *D:* v složce Kafka spusťte následujíce skripty v pořadí, jak jsou uvedené:

- Start Zookeeper
- Start Kafka
- Start Consumer

Toto má za následek, že se spustí instance Apache Kafka spolu s Apache Zookeeper a bude naslouchat na příchozí zprávy.

2. Napojení se do Twitteru a posílaní zpráv do Kafky

Na generování zpráv, které budeme ukládat do Apache Kafka, využijeme možnosti napojení do komunikační platformy Twitter a budeme odtud stahovat aktuální přidané zprávy *(tweets)* a ukládat je do Kafky. Pouze zprávy, které splňují naše filtrační kritérium s klíčovým slovem.

Otevřete si třídu pojmenovanou jako „TwitterToKafka“. Upravte zdrojový kód doplněním následujících prvků:

- Vytvořte objekt typu *string* s hodnotou, která představuje vaše klíčové slovo do vyhledávání. Tento objekt potom přidejte do funkce *track ()* ve spodní časti kódu, který filtruje správy.
- Vytvořte objekt typu *Properties*, který vložíte jako parametr do funkce *KafkaProducer<String,String>().* Definujte v něm parametry podle tabulky:

Tabulka 1 Hodnoty pro funkci Properties

bootstrap.servers	doplňte: IP:Port Kafky
acks	all
retries	0
batch.size	16384
linger.ms	1
buffer.memory	33554432
key.serializer	org.apache.kafka.common.serialization.StringSerializer
value.serializer	org.apache.kafka.common.serialization.StringSerializer

- Doplňte hodnotu názvu vaší kategorie vytvořené v Apache Kafka do objektu *topic*, který je typu *string*

Po vložení potřebných parametrů program spusťte. Jako kontrola správnosti nastavení může posloužit výstup příkazového okna z vašeho Kafka odběratelů *(Consumer)*. Postupně by tam měly přibývat záznamy. Aplikaci v programu Eclipse nechejte běžet.

3. Čtení zpráv z Kafky a ukládaní agregovaných dat do databáze Cassandra

V tomto bodě budete upravovat kód v jazyce Java, ve kterém se napojíte do Apache Kafka, načtete záznamy a uložíte agregované hodnoty do tabulky v databázi Cassandra.

Otevřte třídu *KafkaToCassandra*, kterou následovně upravte:

- V metodě *outputToCassandra* se nacházejí konfigurační parametry na připojení do Cassandra databáze. Doplňte proto IP:Port vaší Cassandra instanci.
- Metoda *main* obsahuje definici tzv. *Dataset<Row>*, která uchovává každý příchozí záznam. Doplňte konfigurační parametry na napojení do Kafka instanci: IP:Port, název kategorie v Kafka.
- *OutputToCassandra* obsahuje objekt typu *String* s názvem q*uery*. Ten určuje SQL dotaz na vložení řádků do databáze. Napište proto jednoduchý SQL příkaz, který vloží každý

řádek z objektu *Dataset<Row>* v tvaru (slovo, počet výskytů) do vámi vytvořené tabulky v databázi Cassandra. Cíl v databázi je potřeba definovat v tvaru *[keyspace.table]*

- Pomocí funkce *printSchema()* zjistěte schéma datasetu *lines.* Jaké údaje dostáváme v Apache Kafka zprávách?
- V definici *Dataset<Row>* se deklaruje možnost *startingOffsets*. Známe dvě základní možnosti nastavení - *earliest* nebo *latest*. Vyzkoušejte obě možnosti. Jaký pozorujete rozdíl?
- Upravte danou proceduru tak, aby se do databáze Cassandra ukládala kromě hodnoty hashtagu i její odstup od začátku *(offset)*.

Tabulka 2 Definice cílové tabulky v Cassandre

String	Integer	String
Hashtag	**Pocet_vyskytov**	**Timestamp**

4. Webové rozhraní pro kontrolu průběhu instance Apache Spark

Na závěr první části zkontrolujte, během průběhu programu, výstup na webovém rozhraní Apache Spark na adrese *http://localhost:4040/* a zjistěte informace jako:

- Kolik úloh *(tasks)* tvoří jeden *Spark Job* a kolik úloh běží v jeden moment
- Počet pracujících uzlú v clusteru
- Kolik běží v jeden moment paralelních dotazů *(queries)*

5. Registrace počítače jako výpočetního uzlu

V druhé části úlohy bude za potřebný registrovat Váš počítač jako výpočetní uzel u hlavního *(master)* uzlu. Od vyučujícího dostanete potřebné údaje jako IP adresa a port. Upravte následující příkaz a spusťte ho pomocí příkazového řádku.

„spark-class org.apache.spark.deploy.worker.Worker spark://<IP>:<PORT>“

Po úspešnom pripojení na master uzel byste mali vidieť nasledujúcu správu:

```
2019-03-04 17:02:35 INFO  Worker:54 - Connecting to master 172.19.60.49:7077...
2019-03-04 17:02:35 INFO  ContextHandler:781 - Started o.s.j.s.ServletContextHandler@2671cfb8{/metrics/json,n
ull,AVAILABLE,@Spark}
2019-03-04 17:02:35 INFO  TransportClientFactory:267 - Successfully created connection to /172.19.60.49:7077
after 69 ms (0 ms spent in bootstraps)
2019-03-04 17:02:35 INFO  Worker:54 - Successfully registered with master spark://172.19.60.49:7077
```

Obr. 5.1 Úspešní připojení k hlavnímu uzlu

6. Nahrávaní dat do clusteru

Upravte program na stahování správ ze sítě Twitter tak, aby se správy ukládali do instance programu Apache Kafka na hlavním uzlu. IP adresu, port a název téma, do které se budou správy logovat Vám sdělí vyučující. Taktéž upravte IP adresu, na které běží instance Apache Cassandra. Program spusťte.

7. Kontrola běhu programu v clusteru

Otevřete si ve webovém prohlížeči stránku v tvare:

http://<IP hlavního uzlu>:8080

Zde můžete vidět základní informace jako:

- Adresa, pod kterou je dostupný Spark na hlavním uzlu
- Počet výpočetních stanic, jejich aktuální stav a dostupný výkon
- Počet běžících aplikací (zde se pro lepší správu clustera zobrazí název aplikace definovaný v zdrojovém programe) a jejích stav
- Po rozkliknuti jednotlivé výpočetního uzlu je možné získat informace o jednotlivých vykonavatelích dané úlohy (executors) na konkrétním uzle. Výpočetní zdroje vykonavatele je možné měnit, stejně jako jejich počet

5.3 Neoptimalizovaný Problém obchodního cestujícího

Cíl úlohy

Základem této laboratorní úlohy je vysvětlení problematiky algoritmu třídy NP-úplný. Laboratorní úloha je sestavená se záměrem seznámit studenta s problematikou Obchodního cestujícího bez optimalizačního algoritmu. To znamená, že s větším počtem měst není tento problém řešitelný v reálném čase.

Zadání

- Nastudujte teorii o problematice obchodního cestujícího
- Naprogramujte kód v jazyku Java tak, aby vypočetl délku trasy pro každou kombinaci měst
- Spusťte úkol s rostoucím počtem měst a pozorujte nárůst náročnosti výpočtu

Teoretický úvod

Problém obchodního cestujícího (dále „TSP“) se popisuje následovně: „Jestliže získáme konečný počet měst a vzdáleností mezi každou dvojicí z nich. Která cesta bude nejkratší, pokud chceme navštívit každé město právě jeden krát a vrátit se zpátky do města, ze kterého jsme cestu započali?“. Tento problém patří mezi nejznámější optimalizační úlohy a řadí se do kategorie NP-úplný. To značí, že pro velké počty měst je problém neřešitelný v reálnem čase a náročnost řešení roste exponenciálně s přibývajícím počtem měst. Již průchodem desítkou různých měst dostáváme přibližně 3,629 miliónů možností průchodů. Navýšením počtu průchodů o pouhé jediné město se počet kombinací zvýší na 39,917 miliónu možností. Co při výpočtu na jednom stroji může znamenat výrazný výpočetní problém.

Mnoho krát se při vysvětlovaní znázorňuje tento problém pomocí obousměrného grafu, kde uzly představuji města a hrany cestu. Každá z hran má přidělené číslo, které reprezentuje např. vzdálenost nebo náklady na cestu. Ukázka takového to grafu je uvedena na obrázku 4.1.

Pracovní postup

Z e-learningu stáhněte soubor TSP_neoptimalizovaný.rar. V něm naleznete před-programované základní časti programu. Třída obsahuje pouze metodu *loadCities*, která má za úlohu načítat seznam měst z JSON souboru.

Doplňte kód tak, abyste pro vybraná města vypočítali každou kombinaci průchodu a do konzoly vypište nejkratší možnou cestu.

5.4 Optimalizovaný Problém obchodného cestujícího řešený v clustery

Cíl úlohy

Úloha navazuje na problematiku obchodního cestujícího z předchozí úlohy. V tomto úkolu je navíc představeno optimalizované řešení s využitím paralelního výpočtu. Cílem úlohy je seznámit studenta s rychlostí výpočtu s nasazením optimalizačního algoritmu. V závěru laboratorního cvičení bude demonstrace výpočtu tohoto optimalizačního problému v clusteru.

Zadání

- Nastudujte teorii o optimalizačním algoritmu
- Doplňte předpřipravený kód v jazyce Java a rozšiřte funkce tak jak popisuje pracovní postup
- Spusťte úkol s různými vstupními proměnnými a pozorujte nárůst náročnosti výpočtu

Teoretický úvod

Teoretický úvod k Problematice obchodního cestujícího byl představený v předcházejícím úkolu. Tento teoretický úvod je zaměřený na objasnění výpočtu s optimalizačním algoritmem.

Výsledek, který nám program poskytne představuje aproximaci k ideální trase. To znamená, že nejlepší trasu, kterou nám program poskytne není reálně nejoptimálnější trasou. Je to z důvodu, že při spuštění je nastaven maximální čas na výpočet. Po jeho uplynutí je získán výsledek, který se považuje za dostatečný. Algoritmus generuje náhodné cesty mezi městy. Aktuálně nejkratší nalezenou trasu zkombinuje s ostatními cestami, aby získal ještě kratší trasy. Lépe algoritmus popisuje obrázek 4.2.

Po startu programu může uživatel zadat několik parametrů, kterými ovlivní běh programu. Konkrétně se jedná o následující parametry:

- **Počet měst** – počet měst, přes které má vypočítaná trasa procházet
- **Maximální doba trvaní iterace** – výpočet každé iterace je ohraničený časem. Po jeho uplynuti se výpočet zastaví
- **Náhodný výběr měst** – příznak, který povoluje programu náhodné vybíraní města ze seznamu
- **Vedoucí *(master)* uzel** – udává IP adresu a port vedoucího uzlu
- **Název aplikace** – název aplikace, která se zobrazí např. ve webovém rozhraní. Slouží k rozlišení aplikace při spuštění různých výpočtů

Jedním z parametrů je i počet měst, které se načítají ze seznamu. První vybrané město se považuje za výchozí i návratové město. V první fázi programu je vygenerován výchozí seznam možných cest. V programu se tento seznam cest označuje jako Generation Zero. Na vstup je přiveden seznam měst, přes které má trasa procházet a funkce vytvoří 100 náhodných možných průchodů. Nejedná se tedy o kompletní seznam, ale pouze o omezený set. Průchod s nejkratší délkou se považuje za základní hodnotu, se kterou jsou později porovnávané nalezené cesty.

Fáze dvě generuje nové cesty pomocí jednoduchého pře násobení. Jedná se o fázi „kombinace“. Každá trasa A je zkombinována s každou jinou alternativní cestou B a to tak, že první polovina trasy A zůstává nezměněná a místo druhé poloviny trasy A je přidána druhá polovina trasy B. Pokud je v druhé polovině trasy B místo, kterým již trasa A prochází, odeberou se zůstávající unikátní města a nově vzniklá cesta je doplněná městy z první poloviny trasy B. Tento proces je znázorněný na obrázku 4.3 v kapitole popisující problém obchodního cestujícího.

Takto nově vytvořená cesta je ještě dodatečně upravena ve třetí fázi. Vygenerují se dvě náhodné pozice a čísla na těchto pozicích se prohodí. Takovýmto náhodným promícháním je zaručeno náhodnější generování nových cest a dosáhnutí dalších kombinací. Pro vetší skupiny měst nemusí dostačovat generovaní cest ve fázi jedna, proto je zavedeno promíchání. Tento způsob je označovaný také jako fáze „mutace“.

V poslední fázi je seznam cest seřazen a je vybrána nejkratší cesta. Postup je opakován dokud není překročen maximální stanovený čas na výpočet nebo není dosaženo maximálního počtu uskutečněných iterací.

Samotná paralelizace výpočtu spočívá v tom, že vytvořený seznam náhodných kombinací průchodů mezi městy je rozdělený mezi uzly. Kolik práce bude daný uzel vypočítávat závisí podle výkonnostních možností každého uzlu. Na příklad, pokud je využit cluster s pěti rovnocennými výpočtovými uzly a výpočet bude proveden pro náhodně vygenerovaných 10 000 cest. V tomto případě bude každý z uzlů počítat vzdálenosti pro přibližně 2 000 cest, jak je patrné z obrázku 4.5. Výstu z každého uzlu je objekt typu Vector<Route>, který je přiveden na vstup funkce pro shromažďování výsledků. Zde se výsledky spoji do jednoho objektu, který se v dalším kroku seřadí podle vypočítané délky. První cesta z tohoto zařazeného seznamu se porovná se zatím nejkratší nalezenou trasou. V případě, že je ujetá vzdálenost menší, nová cesta se nastaví jako nejlepší a program následně pokračuje v dalším hledání dokud nejsou splněné požadavky na počet iterací nebo celkový výpočetní čas.

Pracovní postup

Z e-learningu si stáhněte soubor TSP.rar, který obsahuje JAVA projekt. Následně proveďte následující změny v programu:

- V třídě *Solver* do implementujte funkci *createGenerationZero* tak, aby na vstupu požadovala parametr typu *List<City>*, který představuje seznam měst přes které má

cesta procházet. Pro tento seznam vygenerujte 100 nových náhodných cest s podmínkou aby každá z cest začínala a končila ve stejném městě a každé město je navštíveno právě jeden krát. Tyto cesty nahrajte do proměnné *List<route>*. Tato přeměnná představuje výstup z funkce.

- V třidě *Route* do implementujte funkci *cross* tak, aby pro dva vstupní parametry typu Route vytvořila kombinaci těchto dvou cest následovně: Každá trasa A se zkombinuje s každou jinou alternatívní cestou B a to tak, že první polovina trasy A zůstává nezměněná a přidá se k ní druhá polovina trasy B. Jestliže se již v druhé polovině trasy B nachází město, kterým již trasa A přechází, odeberou se zbylé unikátní města a nově vytvořená cesta se doplní městy z první poloviny trasy B. Pomozte si obrázkem uvedeným v pracovním postupu.

- V té samé třidě definujte tělo funkce *mutate*. Jejím cílem je vyměnit pozice náhodně vybraných dvou měst a tím docílit generovaní nových náhodných cest. Funkce nepracuje se žádnými vstupními parametry a neposkytuje žádný výstup.

- Upravte *main* funkci, kterou naleznete ve třidě *TravellingSalesmanProblem* a definujte parametr, který je možné měnit při volání programu. Dále bude upravovat počet vygenerovaných náhodných cest.

Po úpravách spusťte program pomocí třídy *TravellingSalesmanProblem.* Program zavolejte několik krát po sobě s rozdílnými parametry, tedy pozměňujte počet měst, počet iteraci a maximální doba trvaní iterace. Pozorujte jaký vliv mají tato nastavení na běh programu.

6 VÝKONNOSTNÉ POROVNANIE VÝPOČTU V CLUSTERY

Táto kapitola sa zaoberá porovnaním a analýzou výsledkov, ktoré boli zozbierané pri spustení vyššie popísaného algoritmu Problému obchodného cestujúceho. Cieľom týchto meraní bolo poukázať na výhodnosť, resp. obmedzenia pri spustení algoritmu na viacerých výpočtových zariadeniach súčasne. Zo zvyšných dvoch laboratórnych úloh sa dala pretransformovať do paralelného výpočtu jedine druhá úloha. Tá nie je vo výsledkoch uvedená z dôvodu toho, že pri testových výpočtoch sa vyskytovali rovnaké problémy, ktoré sú popisované nižšie.

Meranie prebiehalo v počítačovej učebni na Ústave telekomunikácií na Fakulte elektrotechniky a komunikačných technológií VUT v Brne. V priebehu merania neboli počítače využívanie na žiadne ďalšie výpočty, ktoré by ovplyvnili výsledky merania. K dispozícií bolo 27 kusov výpočtovej techniky s nasledujúcou konfiguráciou:

- Intel Core i5-6500 3.2GHz 4 jadrá
- 16GB RAM
- Windows 7 Professional

Jeden z počítačov slúžil výhradne ako riadiaci uzol a zvyšných 26 sa postupne pridávalo ako výpočtové zdroje. Pri každom behu programu boli definované rovnaké vstupné parametre programu. Jednalo sa o pozmenenie maximálnej doby behu jednej iterácie na 60 sekúnd a maximálneho počtu iterácií na 4. Na nižšie uvedenej tabuľke číslo 1 sú uvedené hodnoty, ktoré predstavujú dobu trvania výpočtu v sekundách.

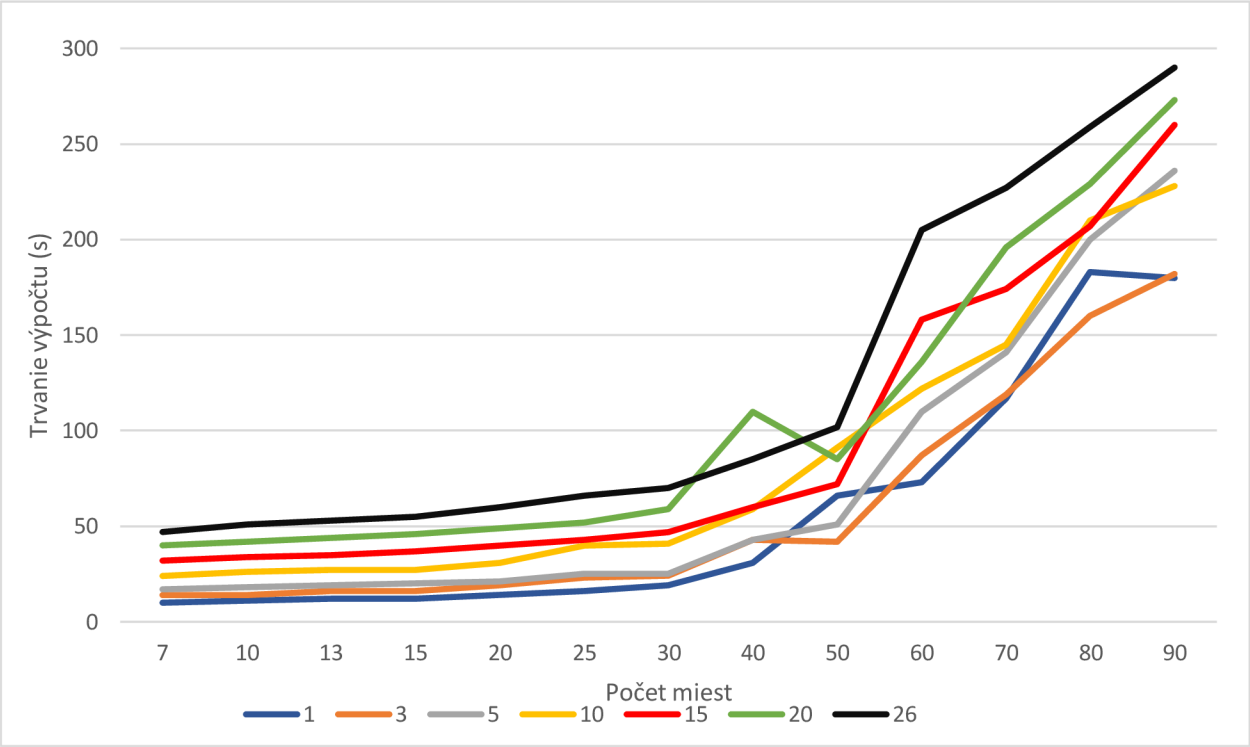
Tabuľka 6.1 Doba behu programu v sekundách

Počet počítačov	Počet miest												
	7[s]	10[s]	13[s]	15[s]	20[s]	25[s]	30[s]	40[s]	50[s]	60[s]	70[s]	80[s]	90[s]
1	10	11	12	12	14	16	19	31	66	73	117	183	180
3	14	14	16	16	19	23	24	43	42	87	119	160	182
5	17	18	19	20	21	25	25	43	51	110	141	200	236
10	24	26	27	27	31	40	41	59	91	122	145	210	228
15	32	34	35	37	40	43	47	60	72	158	174	207	260
20	40	42	44	46	49	52	59	110	85	136	196	229	273
26	47	51	53	55	60	66	70	85	102	205	227	259	290

Rozdiely do merania môže zanášať aj fakt, že pri spustení algoritmu sa optimálna cesta hľadá stále pre tú istú vzorku miest, no v prípade, že sa v nasledujúcej iterácií nájde optimálnejšia trasa, hľadanie pokračuje. Súvisí to s faktom, že prvá generácia ciest, z ktorej sa neskôr vychádza, sa pri viacerých volaniach môže vygenerovať odlišne. Výsledkom teda je, že pri prvom volaní sa optimálna cesta nájde pri 2 iteráciách, v nasledujúcom spustení programu s rovnakými parametrami sa môže objaviť až pri tretej iterácií.

Údaje z tabuľky číslo 1 sú taktiež vynesené do grafu, kde sú rozdiely v meraniach pre každé z miest viac patrné.

Graf 6.1 Porovnanie behu programov s rôznym počtom uzlov



Z grafu číslo 1 je patrné, že pri nasadení vyššieho počtu výpočtových zariadení rastie čas potrebný pre výpočet. Čo ale nie je očakávaným výsledkom. Súvisí to čiastočne s faktom, že pri generovaní týchto dát sa rozdeľovanie záťaže nechalo na automatickom rozdelení podľa uváženia platformy Spark. To ale v praxi znamená, že prvá iterácia hľadania využívala iba 2 jadrá bez ohľadu nato, koľko počítačov mala k dispozícií. Pre vysvetlenie je priložená tabuľka 6.2, ktorá zobrazuje trvanie jednotlivých iterácií v sekundách pri hľadaní optimálnej trasy pre 60 miest.

Tabuľka 6.2 Doba behu programu v sekundách

Počet počítačov	Iterácia		
	1 [s]	2[s]	3[s]
1	43	25	5
3	43	27	7
5	44	40	16
10	43	46	16
15	43	57	35
20	43	75	31
26	43	87	37

Z tabuľky 6.2 je patrné, že prvá iterácia trvá rovnako dlho bez ohľadu na počet dostupných výpočtových zdrojov. Ďalej je viditeľný nárast času v druhej iterácií, ktorý je spôsobený časťou tzv. deserializácie úloh. Tento nárast je dobre viditeľný na obrázku webovom rozhraní Spark

aplikácie. Na prvý pohľad je patrné, že pri spustení úlohy nad 5 počítačmi sa zvyšné počítače, ktoré neboli použité pre výpočet prvej iterácie, potýkajú s problémom deserializácie úloh. Táto časť má za následok zdržanie 4 až 6 sekúnd. Táto fáza sa ale pri výpočte s jedným uzlom nevyskytuje, resp. je minimálna. Samotný čas potrebný na výpočet je už ale nižší v porovnaní s behom nad jedným počítačom. Výpočtový čas s využitím jedného počítača zaberie približne 10 sekúnd, no v prípade paralelného výpočtu s piatimi počítačmi tento čas klesá na 7 sekúnd.

V prípade analýzy tretej iterácie zistíme, že čistý výpočtový čas na jednom počítači trvá približne 3,5 sekundy z celkového 4 sekundového behu iterácie. Zatiaľ čo pri spustení nad piatimi počítačmi výpočtový čas zabral menej ako 3 sekundy. Zvyšných približne 6 sekúnd trvalo riadiacemu uzlu *(driver)* najmä zozbieranie výsledkov z každého z výpočtových uzlu. Apache Spark na svojom webovom rozhraní doporučuje na zníženie enormného času na získanie výsledkov zmenšiť množstvo dát posielaných z každej úlohy vykonávanej na konkrétnom stroji.

Druhá časť výskumnej časti bola zameraná na optimalizovanie behu algoritmu. Zameral som na časť v prvej fáze výpočtu, kedy Spark rozdeľuje úlohy medzi dostupné uzly. Pri ponechaní základného nastavenia sa v každom prípade využili iba 2 jadrá nezávisle na tom, koľko počítačov bolo k dispozícii. Toto správanie sa dá pozmeniť a to tak, že pri nahrávaní úloh cez funkciu *spark-submit* sa dodá konfiguračný parameter *spark.default.parallelism* nastavený na optimalizovaný počet jadier, ktorý je k dispozícii.

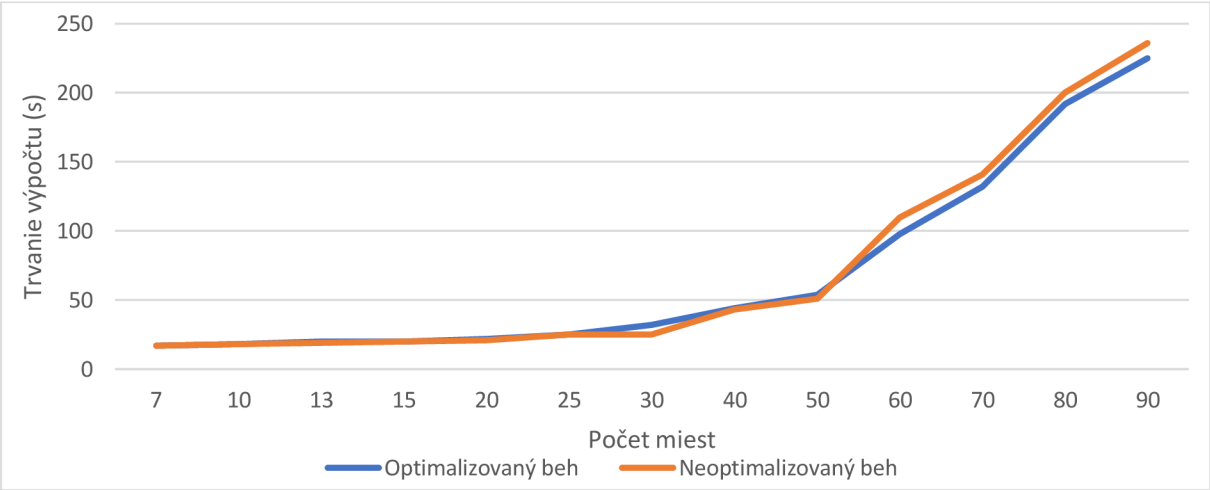
Pri pokusoch sme porovnanie vykonávali nad 5 počítačmi. Nastavil sa preto tento parameter na hodnotu 20. Táto hodnota odpovedá prenásobeniu 5 počítačov a 4 jadier na počítač. Tým dosiahneme deserializáciu iba v jednom kroku a teda nasledujúca iterácia nemusí čakať na zapojenie zvyšných voľných uzlov. Výkonnostný rozdiel je badateľný aj v nasledujúcej tabuľke 6.3. Údaje sú uvedené v sekundách.

Tabuľka 6.3 Porovnanie doby behu programu pred a po optimalizácií

Beh programu	Počet miest												
	7[s]	10[s]	13[s]	15[s]	20[s]	25[s]	30[s]	40[s]	50[s]	60[s]	70[s]	80[s]	90[s]
Opt	17	18	20	20	22	25	32	44	54	98	132	192	225
Neopt	17	18	19	20	21	25	25	43	51	110	141	200	236

Z tabuľky 6.3 možno vypozorovať, že optimalizácia priniesla prínos až pri väčšom počte miest, kde sa úspora výpočtového času pohybuje na úrovni približne 9 sekúnd. Tieto výsledky sú vynesené do grafu 6.2 pre lepšiu názornosť.

Graf 6.2 Porovnanie behu optimalizovaného a neoptimalizovaného algoritmu



Z výsledkov je možné obecne tvrdiť, že výpočet je výhodnejšie vykonávať nad menším počtom uzlov s vyšším výpočtovým výkonom v porovnaní s väčším počtom uzlov, no slabšími výkonovými možnosťami.

Z dôvodu nadmernej deserializácie úloh bola snaha o výraznejšiu transformáciu a optimalizáciu úloh v oblasti globálnych premenných. V podobných úlohách je vo všeobecnosti snaha o vyhnutie sa ich použitiu. Úlohy, ktoré sa rozdistribuovávajú už nenosia v sebe list možných ciest či miest. Snaha v optimalizačnej fáze bola, aby sa premenné predávali priamo na vstup funkcií a výstup z funkcie bol predávaný priamo na vstup inej funkcie bez potreby uloženia výsledkov do globálnej premennej.

Ďalší z návrhov na zlepšenie algoritmu by som videl pri kombinácií výsledkov z viacerých uzlov. Tento krok sa deje v rámci funkcie *combine*. Tu sa zozbierajú zoradené zoznamy ciest podľa dĺžky trasy, kde je preferovaná najkratšia trasa. Podľa môjho návrhu by mal každý z výpočtových uzlov vrátiť iba najlepšiu cestu a nie zoradený zoznam ciest tak, ako je tomu v originálnom algoritme. Týmto sa zmenší veľkosť úlohy, ktorý sa zozbierava z každého uzlu a neskoršie porovnanie naprieč všetkými uzlami sa týmto urýchli. Kedy v prípade 6 výpočtových uzlov je nutné porovnať medzi sebou porovnať iba 6 hodnôt.

Všetky vyššie spomínané optimalizačné kroky spolu s inými, ktoré tu nie sú uvedené žiaľ neviedli k očakávanému zníženiu času na deserializáciu úloh a preto sa ani neuskutočnilo záverečné merania na porovnanie výkonnostnej výhody pri behu na viacerých výpočtových staniciach pomocou optimalizovaného algoritmu.

ZÁVER

Táto diplomová práca sa sústreďovala na priblíženie technológie Apache Structured Streaming. V úvode práce je spomínané nasadenie tohto riešenia v praxi za obdobie posledných niekoľko rokov. Ďalej sú v práci rozpísané open-source technológie ako Apache Kafka, ktorá predstavuje medzivrstvu medzi zdrojom dát a aplikáciou, ktorá ich spracuje. Druhá z technológií je Apache Cassandra- ľahko škálovateľná NoSQL databáza s vysokou dostupnosťou.

Jedným z cieľom tejto práce bolo navrhnúť laboratórne cvičenia do predmetu Paralelné zpracovanie dát na Fakulte elektrotechniky a komunikačných technológií, VUT v Brne. V prvej laboratórnej úlohe si študent prakticky vyskúša spustenie a nastavenie systému Apache Kafka a úvodné nastavenie NoSQL databáze Apache Cassandra.

V druhej laboratórnej úlohe má študent za úlohu doplniť, resp. pozmeniť kód programu, ktorý sa napája cez API rozhranie do spoločnosti Twitter a posiela údaje generované užívateľmi tejto sociálnej siete do študentom definovanej inštancie Apache Kafka. Výsledky sú v programe agregované a nahrávané do NoSQL databáze Apache Cassandra.

Tretia úloha je zameraná na pochopenie problematiky obchodného cestujúceho. V tejto úlohe študenti majú možnosť pozorovať exponenciálny nárast náročnosti výpočtu s pribúdajúcim počtom miest po trase a náplň tejto úlohy spočíva v opäť v doplnení kódu v jazyku Java.

Posledná laboratórna úloha je zameraná na demonštráciu výkonnostnej výhody pri výpočte v clustery. Algoritmus pochádza z voľne dostupných repozitárov a zadanie pre študenta je doplnenie kódu algoritmu za účelom porozumenia princípu prerozdeleniu výpočtu.

Hlavný prínos práce teda predstavujú štyri laboratórne úlohy pre novo vznikajúci predmet Paralelné spracovanie dát na Fakulte elektrotechniky a komunikačných technológií, VUT v Brne. Študentovi majú ukázať možnosti nasadenia open-source technológií pre spracovanie veľkého objemu dát využívaných v praxi.

Posledná časť práce je zameraná na vlastné testovanie algoritmu Problému obchodného cestujúceho. Kapitola obsahuje tabuľky s hodnotami meraní pre varianty s rozdielnym počtom miest a pri postupnom zapojovaní výpočtových zdrojov. Namerané výsledky však neodrážajú očakávaný výstup v podobe dôkazu, že čím viac výpočtových zdrojov zapojíme, tým rýchlejší výpočet bude. Jeden z optimalizačných krokov v podobe nastavenia východzej hodnoty paralelizácie sa prejavil až pri väčšom počte miest v podobe úspory približne 9 sekúnd. V rámci práce bolo dodatočne vykonaných viacero optimalizačných úkonov na zníženie času na deserializáciu úloh, ktorá zaberala najdlhší čas pri spustení. Ani tieto úpravy však nepriniesli požadovaný výsledok.

ZDROJE

[1] Spark Streaming Programming Guide. *Spark Documentation* [online]. Apache Foundation, 2018 [cit. 2018-10-4]. Dostupné z: https://spark.apache.org

[2] B. Zhou et al., "Online Internet traffic monitoring system using spark streaming," in Big Data Mining and Analytics, vol. 1, no. 1, pp. 47-56, March 2018.doi: 10.26599/BDMA.2018.9020005. Dostupné z: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8268735&isnumber=8268 729

[3] W. Li, D. Niu, Y. Liu, S. Liu and B. Li, "Wide-Area Spark Streaming: Automated Routing and Batch Sizing," 2017 IEEE International Conference on Autonomic Computing (ICAC), Columbus, OH, 2017, pp. 33-38. doi: 10.1109/ICAC.2017.36. Dostupné z: http://ieeexplore.ieee.org/stamp/stamp.jsp? tp=&arnumber=8005325&isnumber=8005 289

[4] H. Mushtaq, N. Ahmed and Z. Al-Ars, "Streaming Distributed DNA Sequence Alignment Using Apache Spark," *2017 IEEE 17th International Conference on Bioinformatics and Bioengineering (BIBE)*, Washington, DC, 2017, pp. 188-193. doi:10.1109/BIBE.2017.00-57. Dostupné z: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8251287&isnumber=8251248

[5] D. Choi, S. Song, B. Kim and I. Bae, "Processing Moving Objects and Traffic Events Based on Spark Streaming," *2015 8th International Conference on Disaster Recovery and Business Continuity (DRBC)*, Jeju, 2015, pp. 4-7. doi: 10.1109/DRBC.2015.8. Dostupné z: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7434327&isnumber=7434317

[6] S. Agarwal and B. R. Prasad, "High speed streaming data analysis of web generated log streams," *2015 IEEE 10th International Conference on Industrial and Information Systems (ICIIS)*,Peradeniya, 2015, pp. 413-418. doi: 10.1109/ICIINFS.2015.7399047. Dostupné z: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7399047&isnumber=7398966

[7] Big Data Statistics & Facts for 2017. Waterford Technologies [online]. 2017 [cit. 2018-10-14]. Dostupné z: https://www.waterfordtechnologies.com/big-data-interesting-facts/

[8] Spark Overview. *Spark Documentation* [online]. Apache Foundation, 2018 [cit. 2018-10-14]. Dostupné z: https://spark.apache.org/docs/2.3.2/index.html#spark-overview

[9] Machine Learning Library (MLlib) Guide. *Spark Documentation* [online]. Apache Foundation, 2018 [cit. 2018-10-14]. Dostupné z: https://spark.apache.org/docs/2.3.2/ml-guide.html

[10] GraphX Programming Guide. *Spark Documentation* [online]. Apache Foundation, 2018 [cit. 2018-10-14]. Dostupné z: https://spark.apache.org/docs/2.3.2/graphx-programming-guide.html

[11] HDFS Architecture Guide. *Hadoop Documentation* [online]. Apache Foundation, 2018 [cit. 2018-10-14]. Dostupné z: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[12] Structured Streaming Programming Guide. *Spark Documentation* [online]. Apache Foundation, 2018 [cit. 2018-10-15]. Dostupné z: https://spark.apache.org/docs/2.2.0/ structured -streaming-programming-guide.html

[13] Powered By. *Kafka Documentation* [online]. Apache Foundation, 2018 [cit. 2018-10-20]. Dostupné z: https://kafka.apache.org/powered-by

[14] What is Cassandra? *Cassandra Documentation* [online]. Apache Foundation, 2018 [cit. 2018-11-1]. Dostupné z: http://cassandra.apache.org/

[15] Kafka 2.0 Documentation. *Kafka Documentation* [online]. Apache Foundation, 2018 [cit. 2018-11-1]. Dostupné z: https://kafka.apache.org/documentation/

[16] KONIECZNY, Bartosz. *The role of Apache ZooKeeper in Apache Kafka*. Waiting for code [online]. Bartosz Konieczny blog, 2016 [cit. 2018-11-1]. Dostupné z: https://www.waitingfor code .com/ apache-kafka/the-role-of-apache-zookeeper-in-apache-kafka/read

[17] Cassandra - Quick Guide. *Tutorials Point* [online]. Tutorials Point, 2018 [cit. 2018-11-5]. Dostupné z: https://www.tutorialspoint.com/cassandra/cassandra_quick_guide.htm

[18] Wide column store. *Wikipedia: the free encyclopedia* [online]. San Francisco: Wikimedia Foundation, 2018 [cit. 2018-11-9]. Dostupné z: https://en.wikipedia.org/wiki/Wide_ column_store

[19] Apache Hive TM. *Hive Documentation* [online]. Apache Foundation, 2018 [cit. 2018-11-9]. Dostupné z: https://hive.apache.org/

[20] Apache Storm: Documentation. Apache Storm [online]. Apache Foundation, 2015 [cit. 2019-05-06]. Dostupné z: http://storm.apache.org/releases/1.2.2/index.html

[21] DÄSCHINGER, Melanie. Apache Spark vs. Apache Flink. Woodmark Consulting Blog [online]. 2017 [cit. 2019-05-06]. Dostupné z: https://www.woodmark.de/blog/apache-spark-vs-apache-flink/#fn12

[22] Travelling salesman problem. *Wikipedia: The Free Encyclopedia* [online]. [cit. 2019-05-06]. Dostupné z: https://en.wikipedia.org/wiki/Travelling_salesman_problem

[23] M. Khalil, J. Li, Y. Wang and A. Khan, "Algorithm to solve travel salesman problem efficently," 2016 13th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP), Chengdu, 2016, pp. 123-126. doi: 10.1109/ICCWAMTIP.2016.8079819. Dostupné z: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8079819&isnumber=8079793

[24] Spark of Life: Running a genetic algorithm on Spark. GitHub [online]. 2019 [cit. 2019-05-06]. Dostupné z: https://github.com/nielsutrecht/spark-of-life

SKRATKY

DDoS	Distributed Denial of Service- zahltenie služby požiadavkami z viacero PC
WAN	Wide Area Network
API	Application Programming Interface – rozhranie pre programovanie aplikácií
DB	databáza
TCP	Transmission Control Protocol – sieťový protokol definujúci spôsob výmeny informácií
HDFS	Hadoop Distributed File System – distribuovaný súborový systém
RDD	Resilient Distributed Dataset - súbor dát používaný v Apache Spark
RAM	Random Access Memory – pamäť s náhodným prístupom
NoSQL	NonSQL – nerelačná databáza