FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Architektúra prostredia pre streamové spracovanie veľkých dát
Student:	Maroš Kramár
Vedoucí:	Ing. Jaroslav Kuchař, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Znalostní inženýrství
Katedra:	Katedra aplikované matematiky
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

- Popíšte rozdiely batchového (dávkového) a streamového (prúdového) spracovania veľkých dát a problémy, s ktorými sa môžeme stretnúť pri streamovom spracovaní.
- Preskúmajte a popíšte technológie, ktoré sa používajú pri streamovom spracovaní veľkých dát.
- S využitím existujúcich technológií navrhnite prostredie, ktoré bude umožňovať beh programov určených na streamové spracovanie a analýzu veľkých dát v Hadoop ekosystéme. Programy bežiace v tomto prostredí by mali bez problémov vedieť čítať dáta z Apache Kafka alebo Apache HDFS.
- Zamerajte sa predovšetkým na:
  - distribúciu programov,
  - monitoring,
  - škálovanie podľa záťaže,
  - zotavenie z výpadku,
  - aktualizácie programov bez straty predchádzajúceho stavu.

Ak nájdete viac možných riešení týchto problémov, porovnajte ich a zvoľte najvhodnejšie.
- Vytvorte prototyp, na ktorom bude možné demonštrovať funkčnosť navrhnutého riešenia.

Elektronicky schválil/a Ing. Karel Klouda, Ph.D. dne 3. prosince 2020 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalárska práca

Návrh architektúry prostredia pre streamové spracovanie veľkých dát

Maroš Kramár

Katedra aplikované matematiky

Vedúci práce: Ing. Jaroslav Kuchař, Ph.D.

13. mája 2021

Poďakovanie

Chcel by som poďakovať vedúcemu práce Jaroslavovi Kuchařovi za pomoc a vedenie pri tvorbe práce, taktiež Jakubovi Horníkovi, s ktorým som konzultoval môj návrh a metodiku testovania. Poďakovanie patrí tiež spoločnosti Seznam.cz a.s. za poskytnutie hardvéru pre testovanie.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov. V súlade s ustanovením § 46 odst. 6 tohoto zákona týmto udeľujem bezvýhradné oprávnenie (licenciu) k užívaniu tejto mojej práce, a to vrátane všetkých počítačových programov ktoré sú jej súčasťou alebo prílohou a tiež všetkej ich dokumentácie (ďalej len „Dielo“), a to všetkým osobám, ktoré si prajú Dielo užívať.

Tieto osoby sú oprávnené Dielo používať akýmkoľvek spôsobom, ktorý neznižuje hodnotu Diela, a za akýmkoľvek účelom (vrátane komerčného využitia). Toto oprávnenie je časovo, územne a množstevne neobmedzené. Každá osoba, ktorá využije vyššie uvedenú licenciu, sa však zaväzuje priradiť každému dielu, ktoré vznikne (čo i len čiastočne) na základe Diela, úpravou Diela, spojením Diela s iným dielom, zaradením Diela do diela súborného či zpracovaním Diela (vrátane prekladu), licenciu aspoň vo vyššie uvedenom rozsahu a zároveň sa zaväzuje sprístupniť zdrojový kód takého diela aspoň zrovnateľným spôsobom a v zrovnateľnom rozsahu ako je zprístupnený zdrojový kód Diela.

V Prahe 13. mája 2021 . . . . . .

České vysoké učení technické v Praze

Fakulta informačních technologií

Odkaz na túto prácu

Kramár, Maroš. *Návrh architektúry prostredia pre streamové spracovanie veľkých dát.* Bakalárska práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Spracovanie veľkých dát prežíva v súčastnosti veľký rozmach vďaka internetovým službám a sociálnym médiám. Snaha urýchľovať analýzu a mať k dispozícií výsledky v čo najkratšom možnom čase prináša so sebou nové výzvy a problémy oproti dávkovému prístupu spracovania. Práca prezentuje tieto problémy a naznačuje riešenia, ktoré využívajú moderné open-source technológie pre účel spracovania dát v reálnom čase. Na základe predstavených technológií stavia návrh prostredia, ktoré umožňuje beh analytických programov. Je dôležité, aby fungovanie celého systému bolo spoľahlivé, jednoducho škálovateľné a aby poskytovalo presné výsledky. V tomto prostredí je možné použitie rôznych frameworkov určených pre implementáciu analytických aplikácií, práca sa zaoberá ich porovnaním, meraním a výberom. Na základe návrhu je v práci vytvorený prototyp, ktorý umožňuje spustenie tohto prostredia na osobnom počítači a testovanie funkcionality.

Klíčová slova veľké dáta, distribuované systémy, streamové spracovanie, Apache Hadoop, Apache Kafka, Docker, porovnanie frameworkov

Abstract

In the last decade, data collection and processing is on the rise, thanks to expanding internet services and social media. In effort to speed up this process and obtain results in the shortest time possible, we meet with new challenges and problems compared to traditional batch approach. The thesis presents these issues and indicates solutions, that are used in modern open-source technologies used for data processing in real time. Based on presented technologies, a design of the environment, that allows execution of analytical applications is built. It is important that the whole system is reliable, scalable and can deliver precise results. This environment allows use of multiple frameworks that are compatible, used for implementation of analytic applications. Selected ones are compared from the standpoint of functionality and performance. Based on the design, a prototype is created, that allows running this environment on personal computer for testing purposes.

Keywords big data, distributed systems, stream processing, Apache Hadoop, Apache Kafka, Docker, framework comparison

Obsah

Úvod **1**

1 Streamové spracovanie veľkých dát **3**
1.1 Definícia veľkých dát 3
1.2 Distribuované systémy 4
1.3 Prístupy k spracovaniu dát 4
1.3.1 Batchové spracovanie 4
1.3.2 Streamové spracovanie 5
1.4 Problémy streamového spracovania 5
1.4.1 Transformácie a agregácie 5
1.4.2 Okno, čas vzniku a čas spracovania 6
1.4.3 Stavové spracovanie 7
1.4.4 Garancia doručenia a odolnosť voči výpadkom 7
1.4.5 Distribúcia, spúšťanie programov a škálovanie 9
1.4.6 Aktualizácie 9
1.4.7 Monitoring 9

2 Technológie pre streamové spracovanie **11**
2.1 Apache Hadoop 11
2.1.1 HDFS 12
2.1.2 YARN 13
2.2 Apache Kafka 14
2.2.1 Základné pojmy v Apache Kafka 14
2.3 Apache ZooKeeper 15
2.4 Frameworky pre streamové spracovanie 15
2.4.1 Apache Storm 16
2.4.2 Apache Samza 16
2.4.3 Apache Spark 16
2.4.4 Apache Flink 18

2.5 Prometheus 18
2.6 Grafana 19
2.7 Docker 19
2.7.1 Docker Compose 20

3 Analýza a návrh prostredia **21**
3.1 Funkčné požiadavky 21
3.2 Nefunkčné požiadavky 22
3.3 Vstupný zdroj dát 22
3.4 Spracovanie dát 23
3.5 Spúšťanie a distribúcia aplikácií 23
3.6 Výstup 24
3.7 Monitoring 25
3.8 Schéma 26

4 Výber frameworku aplikačnej vrstvy Hadoop clustra **27**
4.1 Prehľad vlastností 27
4.2 Porovnanie výkonu 28
4.2.1 Testovací program 28
4.2.2 Infraštruktúra pre merania 32
4.2.3 Výsledky merania 33
4.3 Voľba a argumentácia 38

5 Prototyp **39**
5.1 Tvorba prototypu 39
5.2 Konfigurácia a spustenie 41
5.3 Overenie funkčnosti a testovanie 43
5.3.1 Distribúcia a škálovanie 43
5.3.2 Monitoring a alerting 44
5.3.3 Zotavenie z výpadku, aktualizácie 46
5.4 Zoznam webových rozhraní 47
5.5 Príprava pre produkčné nasadenie 48

Záver **49**

Literatúra **51**

A Zoznam použitých skratiek **55**

B Obsah priloženého DVD **57**

Zoznam obrázkov

1.1 Vizualizácia fixného okna [7] 6
1.2 Vizualizácia plávajúceho okna [7] 7

2.1 Architektúra HDFS [10] 12
2.2 YARN v Hadoop ekosystéme [11] 13
2.3 Architektúra monitorovacieho systému Prometheus [26] 19
2.4 Virtuálne stroje versus kontajnery [30] 20

3.1 Navrhnutá architektúra 26

4.1 Diagram infraštruktúry pre merania 33
4.2 Prvé meranie – latencie 34
4.3 Prvé meranie – priepustnosť 34
4.4 Druhé meranie – latencie 34
4.5 Druhé meranie – priepustnosť 35
4.6 Histogram latencií prijatých správ v sekundách 35
4.7 Využitie CPU v percentách 36
4.8 Diskové operácie, čítanie (+) / zápis (–) 36
4.9 Využitie RAM 37
4.10 Prenos dát po sieti, prijaté (+) / odoslané (–) 37

5.1 Web rozhranie systému Prometheus 45
5.2 Pravidlá pre alerting 45
5.3 Grafana dashboard 46

Zoznam tabuliek

4.1 Prehľad vlastností vybraných frameworkov 28
4.2 Zdroje pre testovacie meranie 33

Zoznam výpisov kódu

4.1 Formát vstupných dát 29
4.2 Formát stavu aplikácie 30
4.3 Formát výstupných dát 30
4.4 Mapovacia funkcia testovacej aplikácie v Apache Flink 31
5.1 Dockerfile pre hadoop kontajner 41
5.2 Konfigurácia yarn-site.xml 42
5.3 Časť konfigurácie Apache Flink 42
5.4 Ukážka parametrov pre škálovanie Flink aplikácie 44
5.5 Ukážka Flink akumulátora 44

Úvod

V posledných rokoch môžeme byť svedkami masívneho rozvoja využitia informačných technológií v našich životoch. Vďaka tomu sme dokázali množstvo bežných činností zjednodušiť a zrýchliť. Zaujala nás nejaká téma pri rozhovore a chceme sa o nej dozvedieť viac? Miesto listovania v encyklopédií máme možnosť pomocou internetu vyhľadávať takmer čokoľvek a okamžite. Pri komunikácií s úradmi alebo bankami taktiež miesto návštevy pobočky využijeme dané služby cez internet z pohodlia domova. S tým sa spája nárast ukladaných dát na úrovne, kedy jeden stroj už nieje schopný tieto dáta uchovávať, analyzovať a spracovávať, preto je potrebné úložisko a výpočty distribuovať medzi viacero strojov. Tento princíp sa mimo iných využíva aj v oblasti spracovania veľkých dát.

Jednoduchá dostupnosť výpočtových zdrojov nám umožňuje posúvať analýzu veľkých dát na nové úrovne. V súčastnosti sa stretávame so snahou minimalizovať čas medzi prijatím dát a získaním požadovaných výsledkov z týchto dát. Tento prístup nazývame streamové spracovanie dát. Každá modernizácia ale prináša so sebou aj nové výzvy a problémy, oproti dávkovému spracovaniu je možné pozorovať určité kompromisy za rýchlosť spracovania v presnosti výsledkov.

V súčastnosti máme na výber množstvo technológií, ktoré nám pomáhajú s riešením týchto problémov. Výber správnych technológií a ich prepojenie je kľúčový krok pri návrhu architektúry. Pri spracovaní dát je dôležitá škálovateľnosť, ktorá nám umožní prispôsobovať systém narastajúcim požiadavkám na objem spracovaných a ukladaných dát bez nutnosti návrhu novej architektúry a použitia iných technológií. Táto práca môže pomôcť jednotlivcom ale aj firmám, ktoré chcú modernizovať a vylepšovať svoje analytické systémy tak, aby boli schopné poskytovať okamžité výsledky a zároveň pracovať spoľahlivo.

Cieľom teoretickej časti práce je oboznámiť sa s problémami, ktoré so sebou prináša streamové spracovanie veľkých dát a popísať ich. Ďalším cieľom je preskúmať a predstaviť technológie, ktoré sa v súčastnosti využívajú na tento účel a pomáhajú s riešením uvedených problémov.

Cieľom praktickej časti práce je navrhnúť architektúru prostredia, v ktorom budú bežať programy na analýzu veľkých dát v reálnom alebo skoro reálnom čase, s využitím technológií predstavených v teoretickej časti. Pre návrh je potrebné najskôr definovať požiadavky tak, aby spĺňali moderné štandardy streamového spracovania nezávisle na konkrétnom prípade užitia. Ďalšou časťou je porovnanie frameworkov, slúžiacich pre implementáciu a beh analytických programov na základe stanovených požiadavok a výber pre finálne riešenie. Na základe tohto návrhu potom vytvoriť prototyp, ktorý bude demonštrovať funkčnosť tohto prostredia, vrátane vzorového analytického programu vo vybranom frameworku.

Práca pozostáva z teoretickej a praktickej časti. Do teoretickej časti patrí prvá a druhá kapitola. Prvá kapitola ponúka úvod do spracovania veľkých dát, porovnáva batchový a streamový prístup k spracovaniu dát, popisuje problémy ktoré v streamovom spracovaní vznikajú a načrtne techniky, ktoré sa používajú na ich riešenie. Druhá kapitola sa venuje technológiám, ktoré sú v súčastnosti používané v tejto oblasti a pomáhajú tieto problémy riešiť.

Praktickú časť tvoria kapitoly tri až päť. Tretia kapitola sa zaoberá špecifikáciou požiadavok a návrhom prostredia, ktoré bude umožňovať beh analytických programov pre streamové spracovanie veľkých dát. Popisuje integráciu a spoluprácu jednotlivých technológií. Štvrtá kapitola popisuje postup, ktorý bol použitý pri porovnávaní frameworkov, pre implementáciu programov, ktoré budú spúšťané v tomto prostredí a odhaľuje výsledky porovnania vlastností a výkonu. Posledná piata kapitola sa venuje tvorbe prototypu s využitím popísaných technológií a zvoleného frameworku, a dokazuje, že toto prostredie spĺňa špecifikované požiadavky.

Kapitola 1

Streamové spracovanie veľkých dát

1.1 Definícia veľkých dát

Pojem veľké dáta, často označované aj ako *big data*, nemá jednoznačnú definíciu. Neexistuje konkrétna veľkosť objemu dát, podľa ktorej by sme určili, či už sú dostatočne veľké a môžeme ich nazývať týmto pojmom. Abstraktnejšie sa tento pojem definuje ako dáta, pripadne práca s dátami takého objemu, že je veľmi náročné prípadne až nemožné s nimi pracovať s použitím tradičných techník a technológií, ktoré sa používajú napríklad pri relačných databázach [1]. *Big data* môžeme popísať pomocou tzv. 3V:

- Volume (objem): Nízke ceny diskov nám v súčasnosti umožňujú zbierať a uskladňovať obrovské množstvo dát. Pre väčšie firmy sú to typicky terabajty až petabajty.

- Variety (rozmanitosť): Dáta môžu mať rôznu podobu, štruktúru, môžu byť aj neštruktúrované (napr. obrázky, video). Vyťažovanie znalostí z dát je často problém.

- Velocity (rýchlosť): Na rýchlosť sa vieme pozrieť dvoma rôznymi pohľadmi: rýchlosť, ktorou dáta pribúdajú alebo rýchlosť, ktorou potrebujeme dáta spracovávať. Pri spracovaní rozlišujeme dva prístupy k spracovaniu veľkých dát: streamové (prúdové – v reálnom alebo skoro reálnom čase) a batchové (dávkové – v časových intervaloch). V nasledujúcej časti si tieto prístupy porovnáme a popíšeme podrobnejšie. [2]

1.2 Distribuované systémy

Relačné databázy štandardne bežia na jednom výkonnom stroji. Pri spracovaní veľkých dát tento spôsob ale nie je vhodný – cenovo neefektívny a zároveň veľmi pomalý, prípadne až nefunkčný. S lineárnym nárastom objemu dát, ktoré chceme spracovávať by nám náklady na škálovanie výpočtového výkonu rástli exponenciálne. Po určitej dobe sa ale aj tak dostaneme na hranicu, kde už nie je možné škálovať hardvér vertikálne, teda vylepšovať jeden stroj. V *big data* je preto typické škálovanie výpočtového výkonu do šírky – paralelizácia výpočtov a zároveň distribúcia úložiska medzi viac strojov, ktoré vzájomne komunikujú. Toto zoskupenie strojov nazývame cluster. Pri tomto prístupe nám tak stačia menej výkonné stroje, pri zvyšovaní nárokov cluster rozšírime pridaním ďalších strojov. Stratégia škálovania do šírky sa označuje ako *commodity computing* [3].

Distribúcia nám taktiež umožňuje mať systém, ktorý je odolný voči chybám (tzv. *fault-tolerant*). Dáta sú redundantne ukladané na týchto strojoch, ktoré spolu tvoria vlastný súborový systém. Pri výpadku jedného stroja teda máme prístupné ďalšie kópie rovnakých dát – repliky.

Pri spracovaní každý stroj používa určitú časť dát, ktorú nazývame *partition*. Problém môže nastať vtedy, keď chceme spojiť dáta z rôznych zdrojov, podobne ako v relačných databázach pomocou operácie *join*, pretože sa s vysokou pravdepodobnosťou nachádzajú na rozdielnych strojoch. Pri tejto operácií je potrebné prenášať veľký objem dát po sieti, čo je často pomalé, preto je pri spracovaní veľkých dát snaha minimalizovať počet takýchto operácií. Tento prenos sa často označuje ako *shuffle* [4].

1.3 Prístupy k spracovaniu dát

1.3.1 Batchové spracovanie

Batchový prístup k spracovaniu dát môžeme prirovnať pošte. V priebehu dňa pracovníci pošty zbierajú zásielky od odosielateľov. Na konci dňa sa tieto zásielky roztriedia podľa adresy, na ktorú majú byť doručené a následne ich poštári rozvážajú. Adresáti dostávajú svoje zásielky nasledujúci deň.

V praxi to znamená, že dáta zbierame a spracujeme v pevne daných intervaloch. Ak tieto spracované dáta používame aj na výdaj nejakých informácií (napr. doporučovanie obsahu podľa správania), používame poslednú dávku, ktorá už môže obsahovať neaktuálne informácie. V niektorých prípadoch je teda tento prístup nepoužiteľný. V prípade, že tieto dáta pre výdaj nepotrebujeme mať neustále aktuálne je tento prístup výhodný v tom, že je jednoduchší oproti streamovému spracovaniu na prevádzku a údržbu.

1.3.2 Streamové spracovanie

Druhým prístupom je streamové spracovanie. To by sa zjednodušene dalo prirovnať k emailu, ktorý oproti pošte umožňuje príjemcovi dostať správu a spracovať ju okamžite. V prípade, že sa v nej nachádza nejaká urgentná alebo cenná informácia, nechceme čakať celý deň, ale chceme aby príjemca mohol z tejto informácie profitovať okamžite. Značná výhoda oproti batchovému spracovaniu je teda krátka doba medzi prijatím informácie a jej spracovaním. Toto môžeme docieliť dvoma spôsobmi:

- *Micro-batch stream processing* – založený na batchovom spracovaní, s použitím dávkok, ktoré majú veľmi krátku dobu spracovania (typicky jednotky až desiatky sekúnd).

- *Event stream processing* – označuje sa aj ako *true stream processing*, prichádzajúce dátové body sú spracúvané po jednom. Tento spôsob sa vyznačuje nižšími latenciami, pretože nieje potrebné čakať kým sa naplní predošlá dávka. [6]

1.4 Problémy streamového spracovania

1.4.1 Transformácie a agregácie

Transformácia je proces, pri ktorom upravujeme dátové body nezávisle na iných bodoch z datasetu – množine dátových bodov [6]. Ako príklad môžeme uviesť pričítanie konštanty ku všetkým hodnotám v datasete. Takto upravený dataset nazývame transformovaný dataset. Pri streamovom spracovaní pre nás transformácie nepredstavujú žiadny väčší problém a funguje podobne ako pri batchovom spracovaní, pretože nevyžadujú žiadne dodatočné dáta.

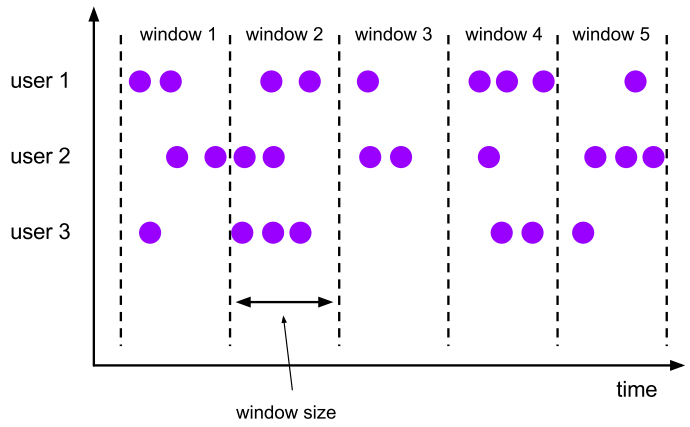
Agregácia je proces, ktorého výsledok pre jeden konkrétny prvok závisí na ostatných prvkoch z množiny nad ktorou túto agregáciu používame [6]. Pri dávkovom spracovaní sa typicky využíva jeden konkrétny dataset, prípadne viac datasetov z minulosti. Príkladom môže byť výpočet priemeru zo všetkých hodnôt, ktoré sa v datasete nachádzajú, prípadne výpočet priemeru na základe kľúča. Avšak pri streamovom spracovaní pracujeme s neohraničenými datasetmi, preto môžeme využiť iba dátové body, ktoré sme už spracovali. Z toho dôvodu si musíme naše dátové body ohraničovať do okien, nad ktorými sa budú tieto agregácie počítať. Tento proces sa často označuje ako *windowing*. Alternatívou môže byť aj využitie aproximačných alebo pravdepodobnostných algoritmov.

1.4.2 Okno, čas vzniku a čas spracovania

Oknom nazývame množinu dátových bodov, ktoré spolu súvisia na základe času. Tento čas môže byť buď čas vzniku dátového bodu (*event time*), prípadne čas, kedy sme tento bod prijali a začali spracovávať (*processing time*). Čas vzniku je generovaný na zariadení kde táto udalosť nastala, je teda priamo súčasťou dátového bodu s ktorým sa posiela k spracovaniu. Rozdiel medzi týmito dvoma časmi je latencia doručenia [6]. Pri použití okna na základe času vzniku udalosti môže nastať problém v prípade, že latencia doručenia je vyššia ako dĺžka okna a dáta nám prídu neskoro. Riešenie tohto problému sa nazýva *late data handling*. Typicky funguje tak, že máme nastavený počet okien, ktoré si držíme v pamäti a v prípade, že nám dorazia neskoré dáta toto okno prepočítame a znovu zapíšeme na výstup. Dáta, ktoré prídu tak neskoro, že nepatria ani do týchto predošlých okien sú typicky ignorované, môže to však spôsobovať nepresnosti vo výsledkoch.

1.4.2.1 Fixné okno

Fixné okno je určené parametrom dĺžky. Každý dátový bod spadá do práve jedného okna, pretože sa okná navzájom neprekrývajú. Každé okno je teda nezávislé od ostatných. Príkladom agregácie nad takýmto oknom môže byť priemer hodnôt za vybranú hodinu.

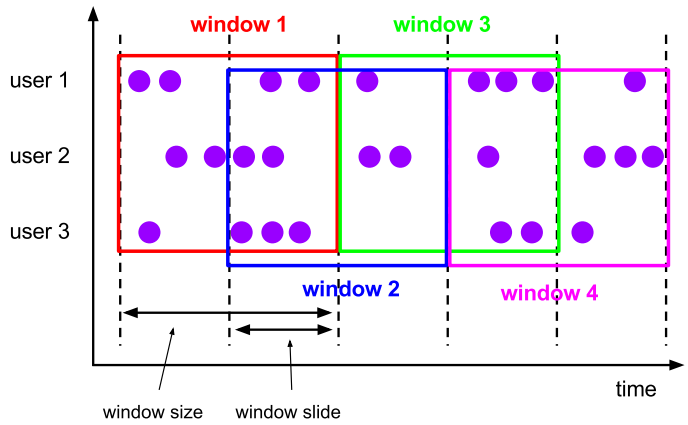


Obr. 1.1: Vizualizácia fixného okna [7]

1.4.2.2 Plávajúce okno

Plávajúce okno závisí od dĺžky okna a frekvencie, ktorá je kratšia ako dĺžka okna. Agregácie sú spúšťané v intervaloch podľa frekvencie a agregujú dátové body, ktoré patria do danej dĺžky okna. To spôsobuje, že sa dátové body môžu

nachádzať vo viacerých oknách súčasne, teda okná sa prekrývajú. Príkladom môže byť priemer hodnôt za poslednú hodinu vypočítaný každých 5 minút.



Obr. 1.2: Vizualizácia plávajúceho okna [7]

1.4.3 Stavové spracovanie

Stavové spracovanie definujeme ako prístup, pri ktorom sa snažíme pochopiť novoprijaté dáta v kontexte dát, ktoré sme prijali už v minulosti [6]. Informácie vyťažené z týchto dát následne použijeme pre aktualizáciu vnútorného stavu. Stav teda môže reprezentovať matematický model, model strojového učenia a pod.

Pre ukladanie stavu sa využívajú rôzne úložiská: databázy, pamäť, disk alebo distribuované úložisko. Pri použití stavu je potrebné dbať na jeho veľkosť. Častým problémom je nárast stavu do objemu väčšieho, ako je jeho úložisko. Preto je dôležitá snaha minimalizovať objem ukladaných dát, prípadne využiť exspiráciu stavu po určitom čase [6].

1.4.4 Garancia doručenia a odolnosť voči výpadkom

Zabezpečiť spoľahlivosť a presonsť výsledkov batchového programu je oproti streamovému prístupu značne jednoduchšie. Ako zdroj dát pri batchovom spracovaní sa typicky používa množina súborov, uložených na distribuovanom súborovom systéme, ktoré využíva replikáciu súborov medzi strojmi ako ochranu proti strate dát v prípade poškodenia niekoľkých strojov. Nastavením replikačného faktora vieme určiť, koľko strojov môže mať výpadok a zároveň dáta boli stále čitateľné. Pri spracovaní dát považujeme zdrojové súbory za tzv. imutabilné, teda nieje možná ich editácia. Vďaka tomu vieme v prípade implementačných chýb alebo hardvérových porúch v čase keď je spustená analýza

výsledok spočítať znova zo zdrojových súborov. To nám garantuje, že každý dátový bod ovplyvní výsledok práve raz [5].

Pri streamovom spracovaní má zmysel zaoberať sa garanciami, ktoré nám určujú, ako sa správa program v prípade neočakávaného výpadku:

- *Exactly-once* – Najsilnejšia garancia, každá správa zo vstupu je práve raz spracovaná alebo zapísaná na výstup.
- *At-least-once* – Každá správa je spracovaná/zapísaná na výstup minimálne raz. Môže nastať situácia, kedy nám vzniknú duplikáty.
- *At-most-once* – Každá správa je spracovaná/zapísaná na výstup maximálne raz. Môže nastať situácia, kedy nám sa nám správa nezapíše. [6]

Rozlišujeme tri miesta, kde je možné určovať garanciu: garancia prečítania správy zo zdroja, garancia spracovania a garancia zápisu výsledku. Aby sme dosiahli *exactly-once* garanciu od vstupu až po výstup, je potrebné aby ju dosahovali všetky tri časti.

V prvom rade musíme zabezpečiť, aby sa nám dáta nestrácali hneď pri zdroji, preto je vhodné použiť zdroje dát, ktoré majú schopnosť retencie dát po určitú dobu. Program, ktorý tieto dáta konzumuje, si zapisuje svoju pozíciu v tomto zdroji (*offset*). V prípade pádu programu je možné pokračovať od bodu, kde skončil. Takýmto zdrojom je napríklad Apache Kafka, bližšie si túto technológiu predstavíme v nasledujúcej kapitole.

Ak využívame stavové spracovanie, zabránenie straty stavu a takisto garanciu spracovania *exactly-once* môžeme dosiahnuť ukladaním pomocných dát na perzistentné úložisko v pravidelných intervaloch. Tento koncept sa nazýva *checkpointing* [5].

Garancia zápisu *exactly-once* je možné vyriešiť idempotentným ukladaním výstupných dát, čo znamená, že viacnásobným zapísaním rovnakého záznamu celkový výsledok neovplyvní, uchováva sa len poslednú hodnota [5].

S použitím týchto techník je teda možné dosahovať *end-to-end exactly-once* garancie aj pri streamovom spracovaní. Nie vždy je to ale požadované, pre zrýchlenie výpočtov sa často používa *at-least-once* garancia v prípadoch kde nám to postačuje a dávame väčší dôraz na výkon aplikácie. Príkladom by mohla byť jednoduchá monitorovacia aplikácia, ktorá nevyžaduje stopercentnú presnosť.

1.4.5 Distribúcia, spúšťanie programov a škálovanie

Pri použití výpočtového clustra je distribúcia programov a rovnako aj sieťová komunikácia medzi jednotlivými uzlami dôležitý problém, ktorý musí riešiť každý, kto chce dáta spracovávať týmto spôsobom. Ak by každý mal implementovať riešenia svojpomocne, k algoritmom ktoré sa venujú samotnej analýze by sa dostal až po veľmi dlhom čase. Preto vznikli frameworky, ktoré umožňujú výpočty spúšťať distribuovane a sprostredkovať komunikáciu medzi uzlami keď je to potrebné.

Pri spúšťaní týchto programov je potrebná správa zdrojov, ktoré využívajú. Na clustri je totiž často spustených mnoho úloh súčasne, preto je potrebné priradiť každému programu správne množstvo zdrojov podľa zvolenej stratégie (rovnomerne, podľa priority, atď). Toto má na starosti tzv. *resource manager*.

Škálovaním rozumieme zvyšovanie výpočtového výkonu pomocou vyššej paralelizácie. Pri spúšťaní programu definujeme, koľko program vyžaduje procesorových jadier a pamäte, o rezervovanie týchto zdrojov a spustenie sa *resource manager* postará. Zároveň je dôležité, aby bolo možné jednoducho škálovať celú architektúru pridávaním nových strojov do clustra.

1.4.6 Aktualizácie

Pri stavovom spracovaní môžeme naraziť na menší problém s aktualizáciami zdrojového kódu našich programov. Je potrebné, aby programy implementovali vytvorenie bodu obnovy, tzv. *savepoint*, čo je súbor, ktorý umožňuje obnovenie stavu programu po jeho zastavení. V prípade, že štruktúra stavu zostáva rovnaká, stačí pri vypnutí starej verzie vytvoriť *savepoint* a pri spustení novej verzie stav obnoviť. V prípade, že potrebujeme pozmeniť štruktúru stavu, napríklad pridaním ďalších hodnôt, ale nechceme prísť o aktuálny stav, je potrebné manuálne implementovať prevod medzi tymito dvoma stavmi, ktorý sa použije pri prvom spustení aktualizovaného kódu [8].

1.4.7 Monitoring

Monitoring je dôležitou súčasťou streamových aplikácií. Slúži nám na sledovanie stavu bežiacich úloh a štatistík o dátach ktoré nimi prechádzajú. Údaje, ktoré sledujeme, nazývame metriky. Tento systém teda zahŕňa technológie, ktoré nám umožňujú metriky zbierať, uchovávať a taktiež technológie, ktoré umožňujú ich vizualizáciu.

Na automatizáciu monitoringu nám slúži tzv. alerting. Slúži na detekciu podozrivých metrík podľa nastavených pravidiel a informuje nás tak o možnom probléme. Pri správnej konfigurácií teda nieje potrebný neustály ľudský dohľad, systém nás o tom informuje automaticky.

Kapitola 2

Technológie pre streamové spracovanie

Z predchádzajúcej kapitoly vieme, že streamové spracovanie napriek svojim výhodám prináša aj mnoho problémov. Keďže v súčastnosti prechádza veľkým rozvojom, vznikajú na trhu rôzne technológie, ktoré tieto problémy pomáhajú riešiť. Ich vhodným prepojením teda dokážeme vytvoriť prostredie, ktoré eliminuje veľkú časť týchto problémov. V tejto kapitole si predstavíme najrozšírenejšie open-source technológie. Keďže je táto oblasť relatívne nová a technológie sa rýchlo vyvíjajú, je pravdepodobné, že v budúcnosti budú obsahovať rozšírenú funkcionalitu, prípadne vzniknú úplne nové technológie, čo môže spôsobiť neaktuálnosť tohto prehľadu.

2.1 Apache Hadoop

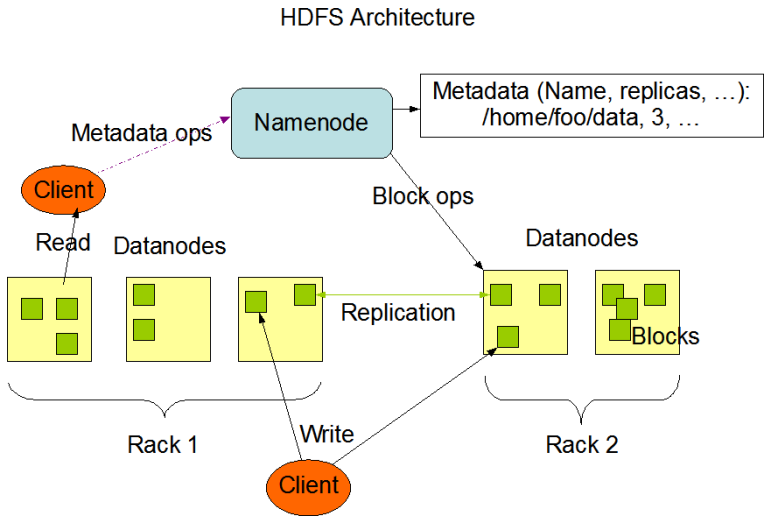
Apache Hadoop je zoskupenie technológií, ktoré umožňujú spoľahlivé a škálovateľné spracovanie veľkých dát pomocou distribuovaných výpočtov. Systém vznikol v roku 2006 a odvtedy sa stal pravdepodobne najpoužívanejším nástrojom v oblasti big data [9]. Kľúčovými súčasťami sú:

- Hadoop HDFS: Hadoop Distributred File System – distribuované úložisko dát
- Hadoop MapReduce: programovací model pre implementáciu aplikácií, ktoré spracúvajú veľké dáta. V súčastnosti ho už pomaly nahradzujú modernejšie a rýchlejšie technológie.
- Hadoop YARN: Yet Another Resource Negotiator – správa a poskytovanie zdrojov clustra pre aplikácie [9]

2.1.1 HDFS

Rovnako ako pri klasických súborových systémoch sú stavebným kameňom HDFS bloky. Rozdielom je veľkosť blokov, ktorá je v tomto prípade oveľa vyššia (predvolená veľkosť je 128 MB) z dôvodu zníženia objemu réžie pri veľkých súboroch. Z tohto dôvodu HDFS nieje vhodný pre ukladanie veľa drobných súborov, réžia v takom prípade môže niekoľkonásobne presiahnuť veľkosť samotných dát [11].

Filozofia HDFS spočíva v prístupe, kde porucha hardvéru je norma a nie výnimka [10]. Výhodou sú teda nízke nároky na spoľahlivosť hardvéru (oproti napr. relačným databázam) vďaka svojej odolnosti voči hardvérovým chybám. S tým je spojená aj nižšia cena jednotlivých strojov. Táto odolnosť je dosiahnutá s využitím redundantnej replikácie blokov naprieč strojmi. V prípade poruchy stroja máme k dispozícií ďalšie repliky, z ktorých je možné dáta prečítať a obnoviť. Poskytuje nám rôzne ďalšie funkcie ako napríklad rovnomernú distribúciu dát naprieč strojmi, dynamická veľkosť blokov v rôznych umiestneniach a iné [11].



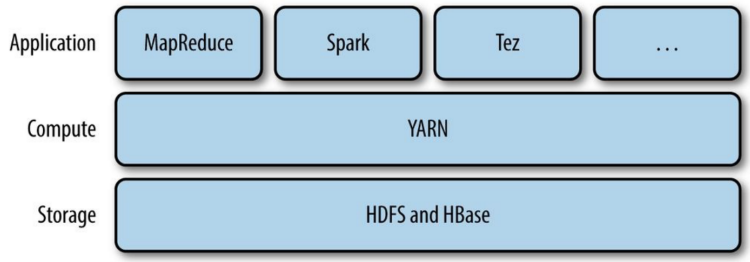
Obr. 2.1: Architektúra HDFS [10]

Stroje patriace do Hadoop clustra fungujú na princípe *master-worker* architektúry:

- *NameNode* (master) – Udržiava metadáta pre všetky súbory a riadi celý súborový systém. Uchováva informácie, ktorý súbor sa nachádza na ktorom DataNode. Pri jeho výpadku stratí celý cluster funkčnosť, preto sa často používa aj záložný NameNode.
- *DataNode* (worker) – Uchováva bloky a na požiadanie ich vydáva. Vo väčšom clustri ich môžu byť tisícky, HDFS ale vie fungovať aj s jedným DataNode uzlom. [10]

2.1.2 YARN

O správu zdrojov v Hadoop prostredí sa stará YARN. Poskytuje rozhranie pre získavanie zdrojov. Toto rozhranie využívajú predovšetkým frameworky vyššej úrovne, ktoré sú s YARN-om kompatibilné (MapReduce, Spark, Flink a iné). Obsahuje rôzne konfigurácie plánovania, ktoré môžu zmeniť výsledné rozdelenie zdrojov. Aplikačný proces, ktorý je spúšťaný na uzloch spolu s pridelenými zdrojmi (CPU, RAM) nazývame kontajner [11]. YARN nám poskytuje aj webové rozhranie, v ktorom môžeme monitorovať využitie clustra, aplikácie ktoré na clustri bežia a tiež prehliadať log súbory z jednotlivých kontajnerov.



Obr. 2.2: YARN v Hadoop ekosystéme [11]

YARN využíva *master-worker* architektúru, rovnako ako HDFS, kde používa nasledovnú terminológiu:

- *ResourceManager* (master) – uzol, ktorý spravuje zdroje v clustri
- *NodeManager* (worker) – uzol, ktorý poskytuje svoje zdroje pre spúšťanie výpočtov [11]

Pri typickom použití beží *ResourceManager* a *NameNode* spolu na jednom alebo dvoch riadiacich uzloch, pričom druhý je záložný, a *DataNode* a *NodeManager* na všetkých výpočtových uzloch.

2.2 Apache Kafka

Apache Kafka je platforma, ktorá zjednodušuje distribuovaný príjem a výdaj dát, ktoré sú označované ako správy. Funguje na princípe *publish/subscribe*, kde publisher (vydavateľ) klasifikuje správy do rôznych tried a subscriber (odberateľ) odoberá správy z určitých tried [12]. Kafku si teda môžeme predstaviť ako centrálny bod, ktorý dokáže prijímať správy od rôznych aplikácií a taktiež rôznym aplikáciam správy poskytovať, preto nieje nutné vyvíjať vlastné systémy pre doručovanie správ medzi aplikáciami. Je navrhnutá tak, aby umožňovala čítanie správ, ktoré už v minulosti boli prečítané (tzv. *replayability*), taktiež poskytuje retenciu správ, teda uchovanie po nastavenú dobu. Pri streamovom spracovaní tak v prípade poruchy žiadne dáta nestratíme, narozdiel od použitia napríklad *TCP/IP sockets*. Zároveň využíva výhody distribúcie pre toleranciu voči poruchám replikáciou uložených správ na viacero uzlov, teda podobne ako v prípade HDFS.

2.2.1 Základné pojmy v Apache Kafka

2.2.1.1 Message

Správu v Kafke definujeme ako pole bajtov, ktoré môže mať ľubovolnú štruktúru (nemusí byť ľudsky čitateľná). Často sa používajú formáty ako JSON (Javascript Object Notation), CSV (Comma-Separated Values) alebo XML (Extensible Markup Language). Taktiež je možné použiť napríklad formát Apache Avro, ktorý sa často používa aj pri batchovom spracovaní a umožňuje kompresiu a jednoduchú deserializáciu vďaka použitiu dátových typov [12].

2.2.1.2 Topic

Správy v Kafke sú rozdelené do topicov. Podľa nich vieme správy triediť a určovať ktorú triedu správ chceme konzumovať v danom programe.

2.2.1.3 Partition

Topic sa delí na viacero partícií, ktoré sú rozmiestnené po strojoch v Kafka clustri. Jedna správa môže byť replikovaná vo viacerých partíciách, vďaka nim nám Kafka zaručuje redundanciu a s tým spojenú prevenciu pred stratou dát.

2.2.1.4 Producer

Producer vytvára správy a zapisuje ich do Kafky. Určuje, do ktorého topicu patrí daná správa.

2.2.1.5 Consumer

Consumer konzumuje správy z jedného alebo viacerých topicov. Správy sú konzumované v presnom poradí, v akom boli zapísané producerom. Zapisuje si pozíciu (*offset*), kde sa aktuálne nachádza, vďaka čomu vie pri výpadku pokračovať tam, kde skončil. Zoskupujú sa do tzv. *consumer groups*, vďaka čomu môže viac konzumentov čítať správy z partícií jedného topicu, ale zároveň každá partícia bude konzumovaná práve raz. Týmto spôsobom je možné škálovať konzumáciu správ [12].

2.2.1.6 Broker

Broker predstavuje jeden uzol (server) v Kafka clustri. Prijíma a ukladá správy, zároveň obsluhuje consumerov.

2.3 Apache ZooKeeper

ZooKeeper je koordinačná služba pre distribuované technológie. Aj pri tejto technológií sa využívajú výhody distribúcie pre škálovateľnosť a odolonosť voči poruchám, takže sa ZooKeeper často používa v clustri. Je nevyhnutný pre beh Apache Kafka, kde sa využíva nasledovne:

- udržiava zoznam aktívnych Kafka brokerov
- určuje leadera (primárnu repliku) pre každú partíciu
- uchováva konfigurácie topicov, počet partícií pre každý topic, lokácie replík, uložené offsety
- dokáže limitovať zápisy producerov a čítania consumerov [13]

2.4 Frameworky pre streamové spracovanie

Pre implementáciu analytických aplikácií sa využívaju frameworky, ktoré za nás riešia časť problémov, ktoré sme popisovali v predošlej kapitole. Programátor sa tak môže sústrediť na implementáciu analytickej časti a nemusí sa príliš zaoberať technickými záležitosťami. Rozhrania nám často umožňujú využívať matematické a štatistické funkcie, ktoré vedia ešte viac uľahčiť implementáciu.

2.4.1 Apache Storm

Apache Storm bol prvým Apache frameworkom, ktorý bol vyvinutý pre streamové spracovanie. Program napísaný v Storm-e je reprezentovaný orientovaným acyklickým grafom, v ktorom poznáme 2 typy vrcholov: *spout*, ktorý je dátovým zdrojom a *bolt*, ktorý reprezentuje transformáciu alebo agregáciu dát. Hrany predstavujú prenos dát. *Bolt* uzly sú rozdistribuované po clustri pre paralelizáciu výpočtov. Dátové body sú v Storm-e reprezentované ako n-tice [14].

Storm nám garantuje *at-least-once* spracovanie dát [14], pričom existuje aj rozšírenie zvané Apache Trident, ktoré dokáže aj *exatcly-once* [15], avšak za cenu nižšieho výkonu. Storm neobsahuje natívnu integráciu s YARN, vyžaduje sa tu rozšírenie Apache Slider [16], čo môže ďalej komplikovať celý systém.

2.4.2 Apache Samza

Apache Samza je framework určený primárne na streamové spracovanie, ale zvláda aj batchové spracovanie. Jeho primárnym zdrojom dát je Apache Kafka, s ktorou je tento framework úzko prepojený. Disponuje jednoduchou integráciou s Hadoop YARN, rovnako dokáže čítať z HDFS, je teda vhodný framework pre použitie v Hadoop Ekosystéme [17].

Umožňuje bezstavové aj stavové spracovanie, pri stavovom spracovaní je garancia stavu *at-least-once* [17], čo môže byť nevýhodou. Je teda vhodnejší skôr pre menej komplexné prípady užitia.

2.4.3 Apache Spark

Apache Spark je nástroj pre spracovanie veľkých dát. Poskytuje rozhranie vyššej úrovne pre jednoduchšiu implementáciu analytických programov. V súčastnosti sa často používa ako náhrada Hadoop MapReduce, pretože dokáže pracovať niekoľkonásobne rýchlejšie [18], ale za cenu väčších nárokov na operačnú pamäť. Prvý prototyp vznikol v roku 2009 na univerzite Berkeley ako výskumný projekt a v roku 2013 bol presunutý do Apache Software Foundation [19]. Ako väčšina frameworkov pre spracovanie veľkých dát je Spark dobre integrovaný s Hadoop ekosystémom, zároveň ale umožňuje beh mimo neho (standalone mode). Bol vytvorený primárne za účelom batchového spracovania, neskôr bol však rozšírený o knižnice, ktoré umožňujú aj streamové spracovanie založené na *micro-batch* princípe. Je implementovaný v jazyku Scala, poskytuje nám rozhrania pre Javu, Scalu, Python a R [20].

Spark obsahuje niekoľko knižníc:

- MLib – knižnica obsahujúca algoritmy pre strojové učenie. Kedže Spark vyniká v iteratívnych výpočtoch, dokáže bežať až 100x rýchlejšie ako Hadoop MapReduce.
- Spark Streaming – umožňuje Sparku fungovať nad neohraničenými datasetmi a vykonávať analýzu v skoro reálnom čase. Pre svoje fungovanie využíva micro-batching, ktorý sme popisovali v prvej kapitole.
- Spark SQL – prináša dátovú abstrakciu nazvanú DataFrame. Rozhranie nad touto štruktúrou obsahuje rôzne štatistické funkcie, zároveň umožňuje spúšťanie SQL dotazov (Structured Query Language), čím zjednodušuje implementáciu.
- GraphX – knižnica grafových algoritmov. [19]

Základom architektúry Apache Spark je tzv. *RDD – Resilient distributed dataset*. Je to kolekcia prvkov, ktorá je imutabilná, distribuovaná medzi uzlami v clustri a odolná voči výpadku uzlov [18, 20]. Umožňuje nám paralelizovať operácie nad datasetom na uzly v clustri. *RDD* môže vzniknúť transformáciou iného *RDD*, paralelizáciou inej kolekcie programovacieho jazyka, alebo načítaním zo súboru. Rozhranie *RDD* je možné použiť priamo, avšak obsahuje iba základné funkcie, preto sa často vyžaduje vlastná implementácia štatistických a matematických funkcii. Nad *RDD* sú postavené pokročilejšie dátové štruktúry *DataFrame* a *DataSet*, ktoré tento problém riešia [21].

Pre streamové spracovanie nám Spark poskytuje dve rozdielne rozhrania, ktoré majú mnoho implementačných a funkčných rozdielov.

2.4.3.1 Spark Streaming

Spark Streaming používa *DStream* rozhranie (*discretized stream*), ktoré je vnútorne reprezentované ako sekvencia *RDD*. Vďaka tomu nám poskytuje *exatly-once* garanciu, za predpokladu použitia vhodného zdroja a výstupu [22].

2.4.3.2 Structured Streaming

Structured Streaming využíva výhody *DataFrame* a *DataSet* rozhrania na ktorom je postavený. Ďalšou výhodou oproti Spark Streaming je schopnosť pracovať s *event time* (čas vzniku udalosti) a práca s oneskorenými dátami (*late-data handling*), ktoré sme popisovali už skôr. Zaručuje nám *exactly-once end-to-end* garanciu. Prináša so sebou tzv. *Continuos Processing* režim, ktorý by mal znížiť latencie na rádovo desiatky milisekúnd, táto funkcia je však

stále experimentálna, dosahuje už iba *at-least-once* garancie a nepodporuje agregácie, preto jej použitie v produkčnom prostredí nieje vhodné a často aj nemožné [23].

2.4.4 Apache Flink

Narozdiel od Apache Spark bol Flink navrhnutý s primárnym účelom streamového spracovania dát, ale je možné použitie aj pre batchové spracovanie. Vznikol na Technickej univerzite v Berlíne v roku 2010, k Apache projektom sa pripojil v roku 2014 [24]. Funguje na princípe reálneho streamového spracovania – spracúva dáta po jednom prvku, na rozdiel od *micro-batch* spôsobu, čo umožňuje dosiahnutie veľmi nízkych latencií. Flink podporuje stavové aj bezstavové spracovanie a poskytuje *exactly-once end-to-end* garancie. Podobne ako Spark Structured Streaming, Flink ponúka všetky pokročilé funkcie streamového spracovania ako napríklad práca s *event time*, *late-data handling*, asynchrónny checkpointing alebo tiež podpora rôznych správcov zdrojov [25].

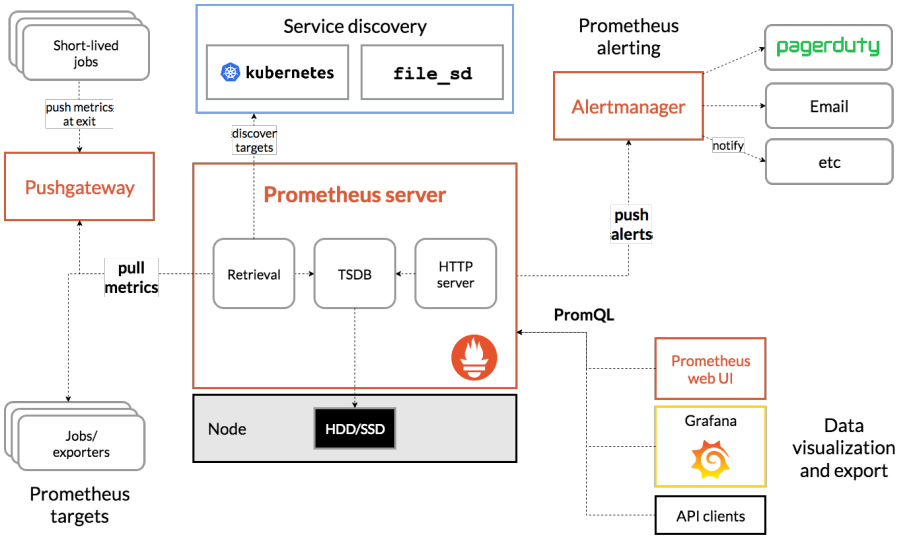
Flink je oproti Apache Spark mladší projekt s menšou komunitou, ktorá ale rýchlym tempom rastie, čo spôsobuje aj rýchly vývoj nových funkcií. Vyniká v jednoduchšom ladení konfigurácie výkonu. Umožňuje výstup jedného operátoru spracovávať ďalšími N operátormi, narozdiel od Spark-u, kde je potrebné tento výstup najprv uložiť do pamäte alebo na disk (*persist*). Na druhú stranu mu chýbajú niektoré základné funkcie, ako napríklad získavanie dát z rôznych súborových formátov (*parsing*), ktoré je potrebné vyriešiť použitím iných knižníc alebo naimplementovať manuálne. Výhodou je tiež integrácia s rôznymi monitorovacími systémami bez nutnosti implementácie vlastných riešení [25].

2.5 Prometheus

Prometheus je pokročilý monitorovací systém. Obsahuje databázu zameranú pre ukladanie časovo závislých hodnôt, ktorá používa vlastný dotazovací jazyk PromQL, ktorý zjednodušuje prácu s týmito dátami a ponúka rôzne agregačné funkcie. Zber dát prebieha primárne v tzv. *pull* režime, teda jednotlivé aplikácie vystavujú svoje metriky na HTTP porte, odkiaľ Prometheus tieto dáta získava v nastavených časových intervaloch. Tento spôsob sa používa pre dlhotrvajúce aplikácie, alebo monitorovanie hardverových metrík strojov. Pre krátkodobé aplikácie existuje súčasť Pushgateway, kam môžu aplikácie poslať svoje metriky, odkiaľ ich Prometheus zozbiera [26].

Pre Prometheus je prvoradá spoľahlivosť a dostupnosť výsledkov na úkor vyššej presnosti. Nemusí byť preto vhodným riešením pre prípady, kde je takáto presnosť dôležitá [26]. Výhodou je jednoduchá integrácia s monitorovacou aplikáciou Grafana, ktorá natívne podporuje Prometheus ako dátový zdroj a taktiež jeho dotazovací jazyk PromQL.

Súčasťou Promethea je aj Alertmanager, ktorý ako už z názvu vyplýva slúži na vytváranie alertovacích pravidiel, na základe ktorých sme potom automaticky informovaný o anomáliach v metrikách.



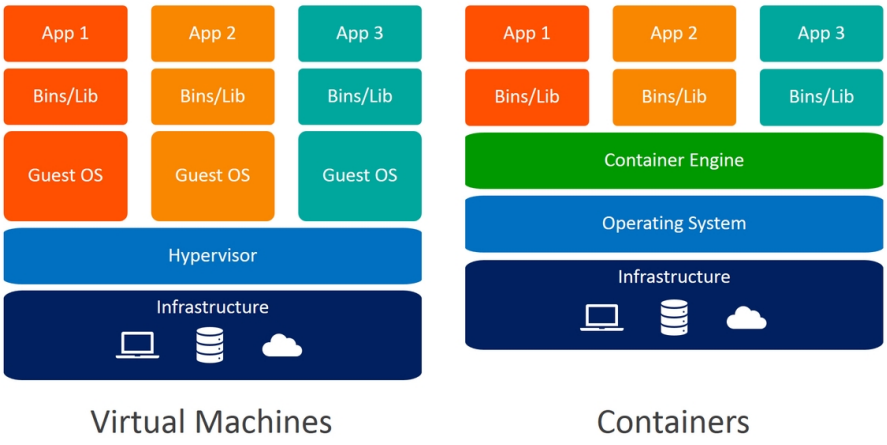
Obr. 2.3: Architektúra monitorovacieho systému Prometheus [26]

2.6 Grafana

Grafana je open-source aplikácia, slúžiaca na vizualizáciu dát. Je výborným nástrojom pre monitoring aplikačných aj hardvérových metrík z dôvodu podpory mnohých dátových zdrojov. Často sa používa v kombinácií s databázou InfluxDB, ktorá patrí medzi tzv. time-series databázy, teda databázy ktoré sú zamerané na uchovávanie hodnôt v čase, prípadne komplexnejším nástrojom Prometheus [28].

2.7 Docker

Docker slúži na vytáranie virtuálnych izolovaných prostredí pre beh aplikácií [29]. Tieto prostredia nazývame kontajnery. Oproti virtuálnym strojom je veľkou výhodou zdieľané jadro operačného systému so strojom hostiteľa, čo výrazne znižuje využitie hardvérových zdrojov. Samotný kontajner teda obsahuje iba binárne súbory a knižnice potrebné pre beh programov, pre ktoré bol vytvorený.



Obr. 2.4: Virtuálne stroje versus kontajnery [30]

Veľkou výhodou je jednoduchá reprodukcia vytvorených kontajnerov pomocou obrazov (*image*), čo je súbor, ktorý obsahuje inštrukcie pre zostavenie kontajnera. Pre reprodukciu výsledkov nám tak stačí jeden konfiguračný súbor s malou veľkosťou na rozdiel od virtuálnych strojov, kde by sme potrebovali celý obraz disku, ktorý môže dosahovať rádovo desiatky gigabajtov. Obraz môže tiež dediť z iných vytvorených obrazov, čo zjednodušuje jeho konfiguráciu. Docker obsahuje vlastnú databázu open-source obrazov, ktorá sa nazýva Docker Hub. Pre mnohé aplikácie vďaka Docker Hub-u môžeme využiť obrazy s pripravenou základnou konfiguráciou od jej samotných vývojárov.

Kontajnery sú vhodné rovnako pre lokálny vývoj a testovanie aplikácií, kde môžu slúžiť ako simulácia reálneho produkčného prostredia, ako aj pre nasadenie aplikácií v produkcií, kde sú do kontajnera zabalené všetky požadované závislosti a konfigurácie spolu s aplikáciou. Aplikácia sa tak priamo spustí v kontajneri, ale dokáže komunikovať s vonkajším prostredím.

2.7.1 Docker Compose

Pre zjednodušenie orchestrácie viacerých Docker kontajnerov slúži Docker Compose. Umožňuje nám pomocou konfiguračného súboru špecifikovať konfigurácie jednotlivých kontajnerov, sprístupnenie HTTP portov pre prenos dát medzi kontajnermi a rôzne ďalšie možnosti. Je teda vhodný v prípade, že chceme dosiahnuť spoluprácu rôznych aplikácií, ktoré sú izolované. Následne je pomocou jedného príkazu možné zostaviť a spustiť celú sieť kontajnerov [31].

Kapitola 3

Analýza a návrh prostredia

Predstavil som technológie, ktoré sú použiteľné pri streamovom spracovaní a riešia mnohé z problémov s ktoré by sa vyskytli pri manuálnej implementácií. Pre správne fungovanie prostredia tieto technológie musia vhodne spolupracovať, samostatne by nedosiahli žiadne výsledky. V tejto kapitole špecifikujem požiadavky, ktoré by malo prostredie spĺňať a podľa nich navrhnem riešenie architektúry. Tieto požiadavky sú prevažne obecné, navrhol som ich tak, aby prostredie zvládalo všetky moderné funkcie, ktoré môžu byť užitočné pri streamovom spracovaní. Prostredie teda nieje zamerané na konkrétny prípad užitia, ale na univerzálnosť a modularitu. Architektúru som rozdelil na nasledujúce časti: vstupný zdroj, spracovanie dát, výstup a monitoring.

3.1 Funkčné požiadavky

1. Streamový vstup: Aplikácie čítajú dáta zo vstupu, ktorý umožňuje konštantný príjem dát a retenciu prijatých dát pre dosiahnutie garancie *exactly-once*.

2. Batchový vstup: Aplikácie pre spracovanie dát vo všeobecnosti často používajú ako vstupné dáta aj výstupy z programov, ktoré dáta spracúvajú batchovo. Je teda potrebné aby bolo možné načítať dáta zo súborov na HDFS.

3. Stavové spracovanie: Aplikácie musia podporovať stavové spracovanie, ktoré sa používa vo väčšine zložitejších prípadov užitia. Stav musí v prípade výpadku dodržovať *exactly-once* garanciu – každým prvkom je ovplyvnený práve raz.

4. Monitoring a alerting: Prostredie má umožňovať jednoduché monitorovanie a vizualizáciu aplikačných metrík v skoro reálnom čase. Na anomálie v metrikách je možné nastaviť upozornenia.

5. Tolerancia voči hardvérovej poruche: Pri výpadku výpočtového uzlu bude bežiaca aplikácia schopný pokračovať vo výpočtoch bez straty dát. Pri výpadku riadiaceho uzlu (systémová chyba, pád Java Virtual Machine) bude aplikácia schopná obnoviť svoju činnosť a pokračovať v spracovaní od pozície kde skončila, hneď ako bude výpadok opravený.

6. Aktualizácie aplikácií bez straty stavu a predchádzajúcej pozície na vstupe : Aplikácie si ukladajú stav a pozíciu už spracovaných správ zo vstupu do spoľahlivého úložiska - vedia potom pokračovať z tejto pozície po aktualizácii.

3.2 Nefunkčné požiadavky

1. Škálovateľnosť: Pri zvyšujúcich sa nárkoch na výkon bude možné programy jednoducho škálovať, teda prideliť im viac zdrojov.

2. Priepustnosť: Aplikácie bežiace v tomto prostredí budú schopné spracovávať dáta v skoro reálnom čase.

3. Využiť nekomerčné technológie: Prostredie musí byť postavené na open-source technológiách.

4. Kompatibilita s Hadoop ekosystémom: Aplikácie využívajú YARN pre správu zdrojov a HDFS ako perzistentné úložisko. Hadoop ekosystém sa často používa aj pri batchovom spracovaní, preto môže byť celý systém postavený na jednom základe.

3.3 Vstupný zdroj dát

Ako zdroj dát pre analytické programy v tejto architektúre som zvolil Apache Kafka, čo je v súčastnosti pravdepodobne najpoužívanejšia technológia pre tento účel. Disponuje všetkými dôležitými funkciami pre zabezpečenie nízkych latencií a zároveň je dobre podporovaná aj v mnohých frameworkoch pre spracovanie dát, ktoré obsahujú vstavané rozhrania pre komunikáciu s Apache Kafka.

Alternatívou by mohol byť napríklad Apache Pulsar, ktorý už od základu podporuje pokročilejšiu funkcionalitu, ako napríklad geografickú replikáciu celých clustrov. Na rozdiel od Apache Kafka má ale táto technológia menšiu

komunitu a rovnako menšiu podporu frameworkov pre spracovanie dát, pričom funkcie, ktoré poskytuje Kafka pre toto prostredie postačujú.

Kafka pre svoje fungovanie vyžaduje Apache ZooKeeper. Z dôvodu odolnosti voči hardvérovým poruchám bude potrebné, aby tieto technológie bežali na viacerých strojoch. Pri malých clustroch je možné aby obidve technológie bežali na každom stroji súčasne, čím by sme znížili počet strojov a rovnako aj náklady. Z dôvodu škálovania je ale výhodnejšie mať samostatný cluster pre Kafku a ZooKeeper, čo so sebou prináša aj výhody vyššej stability, dostupnosti a redundancie. Obe aplikácie často zapisujú a čítajú z disku, preto vďaka tomuto bude benefitom aj vyšší výkon. Keďže ZooKeeper využíva majoritné hlasovanie, je dôležité aby týchto strojov bol nepárny počet.

3.4 Spracovanie dát

Základom časti pre spracovanie dát je Hadoop ekosystém. Ak sa na tento ekosystém pozrieme z pohľadu vrstiev (obrázok 2.2), na najnižšej vrstve ktorá slúži ako úložisko budeme využívať HDFS. Pri streamovom spracovaní slúži primárne na ukladanie stavu programov – *checkpointing* a vytváranie *savepoint*-ov, ale aj v prípade, ak by program vyžadoval ako vstup dáta, ktoré sú výsledkom batchového spracovania.

Na výpočetnej vrstve bude YARN, ktorý bude slúžiť pre správu zdrojov aktuálne bežiacich úloh, vďaka čomu je možné aplikácie bežiace v tomto prostredí jednoducho škálovať. Po počiatočnej konfigurácií týchto dvoch vrstiev by nemalo byť potrebné robiť žiadne väčšie zásahy pri zmene frameworku aplikačnej vrstvy.

Na aplikačnej vrstve tejto architektúry budú bežať programy vytvorené za pomoci zvoleného frameworku. Architektúra je nezávislá od použitého frameworku (za podmienky, že framework poskytuje integráciu s Hadoop-om), takže umožňuje aj beh viacerých programov založených na rôznych frameworkoch zároveň. Je možné použiť ľubovoľný z frameworkov zmienených v druhej kapitole. Výber frameworku je kľúčová časť k efektívnemu spracovávaniu veľkých dát, či už z hľadiska výkonu alebo pohodlia implementácie, konfigurácie, testovania a nasadenia. Porovnaním a výberom jedného konkrétneho frameworku podľa špecifikovaných požiadavok sa zaoberá nasledujúca kapitola.

3.5 Spúšťanie a distribúcia aplikácií

Ak by sa k spúšťaniu programov používal master uzol Hadoop clustra, mohlo by ľahko dôjsť k zmenám v konfigurácií, ktoré ovplyvňujú celý cluster. Preto je výhodnejšie mať samostatný stroj, ktorý slúži len na tento účel, kvôli vyššej

bezpečnosti a lepšej izolácií, prípadne spúšťať aplikáciu v izolovanom kontajneri. Naskytujú sa tieto riešenia:

- Distribúcia aplikácií pomocou vytvárania aplikačných balíkov (napríklad .deb), ktoré sa priamo nainštalujú na launcher. Nevýhodou je neexistujúca izolácia aplikácií a s tým povinnosť využívania spoločných verzií závislostí s inými balíkmi. Problémom tiež môže byť zložitejšie nasadzovanie a správa konfiguračných nástrojov.

- Distribúcia aplikácií pomocou Docker kontajnerov, ktoré sú spúšťané na launcheri. Veľkou výhodou je izolácia a z toho vyplývajúca možnosť použitia rozdielnych verzií frameworkov na jednom clustri, jednoduchšia práca pri nasadení a vyššia bezpečnosť. Nevýhodou môže byť zložitejšia počiatočná práca pri vytváraní obrazov. Toto riešenie môže spôsobovať problémy pri použití virtualizovaného clustra z hľadiska bezpečnosti alebo výkonu.

- Distribúcia aplikácií pomocou Docker kontajnerov spúšťaných pomocou externého orchestračného nástroja (napr. Kubernetes) ponúka výhody predošlého riešenia a eliminuje problém s virtualizovaným clustrom. Zároveň ale prináša ďalšiu komplexitu, nutnosť konfigurácie a údržby.

Ja som pre prostredie zvolil druhú možnosť, ktorá je dobrým kompromisom medzi výhodami a pridanou komplexitou. V prípade potreby je možné rozšírenie na tretiu možnosť, ktorá umožňuje používať rovnaké, už existujúce kontajnery.

3.6 Výstup

Pre zaistenie *exactly-once* garancie je potrebné použiť výstupnú databázu, do ktorej je možné zapisovať dáta idempotentne. Vhodnými databázami by mohli byť napríklad Apache HBase, MongoDB, Couchbase a iné.

Ak výstupy chceme znovu použiť v ďalších programoch a nepotrebujeme nekonečnú perzistenciu dát, vhodným výstupom by mohla byť rovnako ako pri vstupe Apache Kafka. Pri tomto výstupe nieje možné použiť ľubovoľný framework, nie všetky totiž disponujú mechanizmami pre zabezpečenie *exactly-once* garancie. Tu vyniká Apache Flink, ktorý disponuje *two-phase commit* funkciou, ktorá umožňuje dosahovať *exactly-once* pri integrácií s Apache Kafka [32].

V prípade tohto projektu pre jednoduchosť postačí Kafka, ktorú už používame ako zdroj dát, ušetríme tak zbytočnú komplexitu pri prototype. Pre zložitejšie prípady užitia bude potrebný vhodný výber inej výstupnej databázy.

3.7 Monitoring

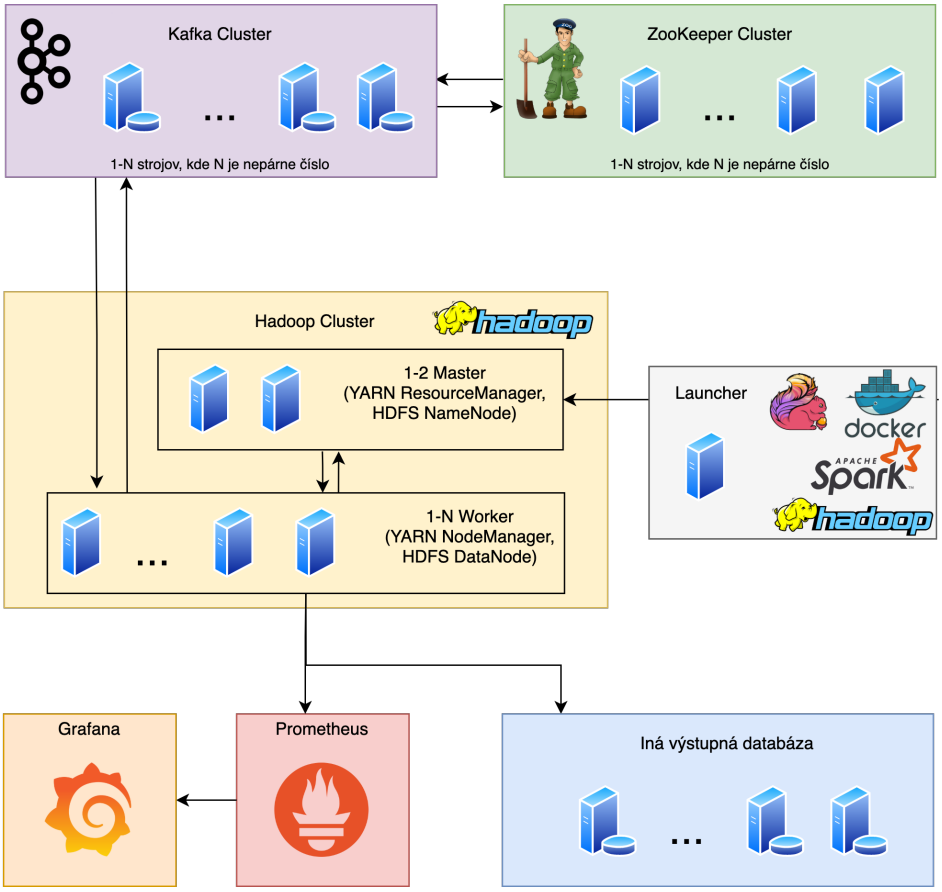
Pre monitorovaciu infraštruktúru streamových systémov je ideálnym riešením Prometheus. Zvolil som ho z dôvodu, že obsahuje množstvo funkcií, narozdiel od jednoduchých databáz pre časové rady – zabudovaný systém pre vyhodnocovanie a posielanie alertov, agregačné funkcie, integrácia so systémom Grafana. Taktiež obsahuje rôzne rozšírenia, napríklad pre kontrolu živosti HTTP portov, Node Exporter, ktorý slúži na získavanie hardvérových metrík a ďalšie. Tu znova vyzdvihnem Flink, pretože vie spolupracovať s Prometheom, po konfigurácií je možné automaticky zasielať aplikačné metriky, pri iných frameworkoch je potrebné vymyslieť vlastné riešenie.

Alternatívou by bolo použitie databáz zameraných na ukladanie dát s časovými radami (napr. InfluxDB), ktoré ale vo väčšine prípadov obsahuje iba základné funkcie a prepojenie s analytickými aplikáciami bude často potrebné riešiť svojpomocne. Toto riešenie by som volil iba v prípade, že je potrebná vlastná implementácia nejakých špecifických funkcionalít, v iných prípadoch to neprinesie žiadne výhody.

Pre vizualizácie som zvolil už zmienený systém Grafana, ktorý je možné jednoducho napojiť na Prometheus server a dotazovať sa na dáta pomocou jazyka PromQL. V základe obsahuje množstvo vizualizácií, ktoré sú ďalej rozšíriteľné. Zároveň je jednoducho prepojiteľný aj s inými databázami a systémami.

3.8 Schéma

Na nasledujúcej schéme sú zobrazené jednotlivé servery podľa môjho návrhu, spolu s technológiami, ktoré na nich bežia. Šípky označujú smer, v ktorom sa pohybujú dáta.



Obr. 3.1: Navrhnutá architektúra

Kapitola 4

Výber frameworku aplikačnej vrstvy Hadoop clustra

Predstavil som návrh architektúry, ktorý je použiteľný pre streamové spracovanie dát s vhodným frameworkom, pomocou ktorého sa budú vytvárať programy. Architektúra je modulárna, teda je možné použiť rovnaký základ (Apache Hadoop) pre rôzne frameworky, ktoré ho podporujú. V tejto kapitole frameworky porovnám z viacerých hľadísk a vyberiem jeden, ktorý považujem za najvhodnejší podľa požiadavok popísaných v predošlej kapitole.

4.1 Prehľad vlastností

V prehľade som sa zameral na vlastnosti vyplývajúce z požiadavok. Môžeme tu vidieť, že Storm nepodporuje stavové spracovanie (bez rozšírenia Trident), Samza dosahuje iba *at-least-once* garanciu stavu. Spark Streaming a Spark Structured Streaming sú veľmi podobné, druhý zmienený však obsahuje pokročilejšie a modernejšie rozhranie s rozšírenejšou funkcionalitou. Rozhodol som sa teda ďalej porovnávať už iba 2 z týchto frameworkov: Flink a Spark Structured Streaming, ďalej označovaný iba ako Spark.

	Storm	Samza	Spark Streaming	Structured Streaming	Flink
Podpora jazykov	ľubovoľný	Java	Java, Scala, Python, R	Java, Scala, Python, R	Java, Scala, Python
Integrácia s resource managermi	YARN za pomoci Apache Slider	YARN	YARN, Mesos	YARN, Mesos	YARN, Mesos, Kubernetes
Garancia stavu	nepodporuje stav	at-least-once	exactly-once	exactly-once	exactly-once
Integrácia s Apache Kafka	áno	áno	áno	áno	áno
Integrácia s HDFS	áno	áno	áno	áno	áno
Vlastná implementácia vstupu a výstupu	áno	áno	áno	áno	áno

Tabuľka 4.1: Prehľad vlastností vybraných frameworkov

4.2 Porovnanie výkonu

Dôležitým aspektom pri programoch pre streamové spracovanie je aj efektívnosť využitia zdrojov, keďže programy musia bežať nepretržite. Z pohľadu využitia hardvéru nás zaujíma hlavne využitie procesora, pamäte RAM, prenos dát po sieti a diskové operácie. Pri spracovaní dát nás zase zaujíma počet dátových bodov ktoré je schopný program spracovať za jednotku času – priepustnosť, a takisto aj časový rozdiel medzi vytvorením dátového bodu a jeho spracovaním a zapísaním výsledku – latencia.

4.2.1 Testovací program

Pri meraní som sa zameral na stavové spracovanie, ktoré sa používa skoro pri každom komplexnejšom programe a vyžaduje viac zdrojov ako keby sme používali jednoduché transformácie. Príkladom stavového spracovania je detekcia anomálií podľa kľúča. Účelom detekcie anomálií je určovanie hodnôt, ktoré sa značne odlišujú od ostatných hodnôt z rovnakej množiny. Zameriam sa na popis implementácie algoritmu detekcie, funkcie pre načítanie dát zo zdroja alebo zápis sú veľmi podobné, preto sa nimi nebudem v tomto texte zaoberať. Zdrojový kód oboch programov je súčasťou prílohy práce.

Pre vstupné dáta som zvolil nasledovný formát:

```
case class Event(id: Long, eventTimestamp: Long, value: Long)
```

Výpis kódu 4.1: Formát vstupných dát

- id: jednoznačný identifikátor (kľuč) vo formáte Long
- eventTimestamp: časová známka vytvorenia danej správy vo formáte unix timestamp – počet sekúnd od 1.1.1970
- value: náhodne generovaná hodnota z normálneho rozdelenia

Na detekciu anomálií môžu slúžiť rôzne pokročilé metódy založené na strojovom učení, ja som sa však rozhodol zvoliť jednoduchšiu variantu, ktorá na účely merania výkonu plne postačuje. Je založená na metrike zvanej *z-score* alebo *štandardizované skóre* [33]. Za pomoci výberovej smerodajnej odchýlky a priemeru je možné anomálie určovať na základe nasledujúcej nerovnice:

$$|x_n - \bar{x}_n| > c * s_n$$

kde x je množina reálnych čísel reprezentujúca prijaté hodnoty,
x_n je posledná prijatá hodnota, ktorú chceme testovať,
$\bar{x}_n = \frac{\sum_{i=1}^{n} x_i}{n}$ je aritmetický priemer množiny, ktorej posledný prvok je x_n,
c je vhodne zvolená konštanta, ktorá ovplyvňuje citlivosť detekcie,
$s_n = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$ je smerodajná odchýlka.

Ak uvedená nerovnica platí, potom označujeme hodnotu x_n ako anomáliu.

Pri klasickom výpočte priemeru a smerodajnej odchýlky by do stavu pre každý kľuč bolo potrebné ukladať všetky prvky, pri prijatí nového prvku hodnoty znovu prepočítavať. Pre zrýchlenie a zníženie veľkosti stavu je možné použiť inkrementálny výpočet [27].

$$s_n = \sqrt{\frac{d_n^2}{n-1}}$$

$$d_n^2 = \sum_{i=1}^{n}(x_i - \bar{x}_n)^2$$

Hodnota d_n^2 reprezentuje kvadratický priemer odchýliek od strednej hodnoty (v tomto prípade aritmetický priemer). Túto hodnotu je možné spočítať inkrementálne, bez znalosti celej množiny hodnôt:

$$d_n^2 = d_{n-1}^2 + (x_n - \bar{x}_n) * (x_n - \bar{x}_{n-1})$$

$$\bar{x}_n = \bar{x}_{n-1} + \frac{x_n - \bar{x}_{n-1}}{n}$$

Pri použití tohto spôsobu výpočtu bude stav mať nasledujúci formát:

```
case class EventStats(count: Long, avg: Double, dSquared: Double)
```

Výpis kódu 4.2: Formát stavu aplikácie

Na základe tohto stavu vieme pri ľubovolnom prichádzajúcom prvku spočítať požadované hodnoty pre detekciu. Pre presnosť výpočtov je vhodné stanoviť hranicu počtu prvkov, kedy chceme začať detekciu. Výstupom budú dáta nasledovného formátu:

```
case class Result(id: Long, eventTimestamp: Long, avg: Double, std:
    Double, flag: String)
```

Výpis kódu 4.3: Formát výstupných dát

- id: jednoznačný identifikátor (kľuč) vo formáte Long
- eventTimestamp: časová známka vytvorenia danej správy vo formáte unix timestamp – počet sekúnd od 1.1.1970, vo výstupe ju používam na meranie odozvy
- avg: priemer z doteraz prijatých hodnôt
- std: smerodajná odchýlka z doteraz prijatých hodnôt
- flag: určuje či ide o anomáliu, prípadne informuje o nedostatku dát pre daný kľuč, možné hodnoty sú: `"outlier"`, `"ok"` alebo `"incomplete_data"`

Na základe týchto znalostí je možné implementovať funkciu, ktorá bude počítať požadované hodnoty pri prijatí každého prvku. Táto funkcia funguje ako tzv. mapovacia funkcia, teda funkcia ktorá pre každý prvok na vstupe vykoná nejakú akciu (transformácia alebo agregácia), pri ktorej vznikne výstup.

```
private var statsState: ValueState[EventStats] = _

override def flatMap(input: Event, out: Collector[Result]): Unit = {

  val tmpStats = statsState.value
  val stats = if (tmpStats != null) tmpStats else EventStats(0, 0,
      0)

  val updatedCount = 1 + stats.count
  val updatedAvg = stats.avg + (input.value - stats.avg) /
      updatedCount
  val updatedDSquared = stats.dSquared + (input.value - updatedAvg)
      * (input.value - stats.avg)

  val updatedStats = EventStats(updatedCount, updatedAvg,
      updatedDSquared)

  val result = if (updatedStats.count >= 10) {

    val variance = updatedStats.dSquared / updatedStats.count
    val std = math.sqrt(variance)

    val flag = if (input.value > updatedStats.avg + 2 * std
               || input.value < updatedStats.avg - 2 * std)
                   "outlier" else "ok"

    Result(input.id, input.eventTimestamp, updatedStats.avg, std,
        flag)
  } else {
    Result(input.id, input.eventTimestamp, 0, 0, "incomplete_data")
  }

  out.collect(result)
  statsState.update(updatedStats)
}
```

Výpis kódu 4.4: Mapovacia funkcia testovacej aplikácie v Apache Flink

V Apache Spark bude mapovacia funkcia veľmi podobná, rozdiel je iba v signatúre funkcie, kde miesto jedného *Event* dostávame *Iterator*, teda odkaz na množinu prvkov, pretože Spark funguje na *micro-batch* princípe. Výstupom funkcie je tým pádom tiež *Iterator*. Pri používaní podobných funkcií je potrebné dbať na to, že veľkosť stavu nemôže byť neobmedzená. Je možné, že na vstupe by nám prišlo veľmi veľa kľučov, ktoré by sa už do pamäte nezmestili, preto je potrebná ich exspirácia. V tomto prípade mám ale pri generovaní obmedzený počet možných kľúčov, preto to nieje potrebné.

Aby program mohol fungovať, je potrebné ešte definovať serializáciu a deserializáciu dát z/do Apache Kafka. Ja som zvolil formát CSV, kde sú jednotlivé hodnoty oddelené pomocou nastaveného oddeľovača, v mojom prípade bodkočiarka. Vstupné dáta po dekódovaní z poľa bajtov na reťazec vyzerajú nasledovne:

```
0;341
11;6743
42;1763
```

Výstupné dáta vyzerajú nasledovne:

```
0;1619366088231;0;0;"incomplete_data"
11;1619366088342;5332.342345456;2189.889796431;"ok"
42;1619366088433;6399.615635762;1832.203230063;"outlier"
```

4.2.2 Infraštruktúra pre merania

Pre generovanie a zápis správ v požadovanom formáte do Apache Kafka som vytvoril jednodchý producer v jazyku Python, ktorý je súčasťou prílohy práce. Časová známka sa automaticky vytvorí pri zápise, ukladá sa do metadát správy, z ktorej sa pri deserializácií extrahuje. V tomto prípade má veľký význam, pretože je vďaka nej možné merať latenciu od vzniku správy až po zápis spracovaného výsledku. Pre toto meranie používam ďalší skript v jazyku Python, ktorý plní úlohu Kafka consumera. Číta správy z výstupného topicu a určuje latenciu každej správy odčítaním časovej známky vytvorenia správy od časovej známky zápisu výsledku. Priepustnosť je merateľná monitorovaním počtu správ, ktoré tento consumer prečíta za sekundu.

Na monitorovanie týchto hodnôt som využil už skôr zmienený Prometheus, ktorý je použitý aj v návrhu architektúry. Consumer metriky vystavuje na HTTP porte, cez ktorý Prometheus tieto hodnoty zbiera (tzv. *scraping*) a ukladá do databázy. Na meranie metrík hardvéru som využil Prometheus Node Exporter. Pomocou Grafany je potom možné dotazovať Prometheus a hodnoty vizualizovať.

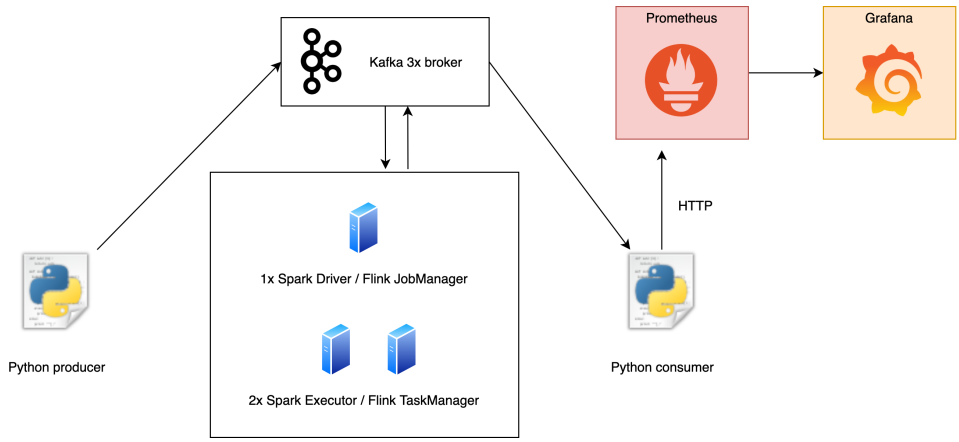
Zmienené skripty využívajú open-source knižnice *prometheus-client*, *kafka-python* a *numpy*, ktoré sú voľne prístupné v oficiálnom repozitári Python balíčkov na adrese `https://pypi.org/`.

Merania som spúšťal na virtuálnom clustri, ktorý bol vyhradený pre tieto experimenty, výkon teda nebol ovplyvnený inými aplikáciami. Kafka cluster bol zdieľaný aj pre iné účely, poskytuje však dostatok zdrojov, preto by nemal výrazne ovplyvniť namerané výsledky. Hodnoty zdrojov som volil tak, aby

odpovedali prípadu užitia a počtu generovaných sprav – priemerne 50 000 správ za sekundu. Program som testoval s rôznymi nastaveniami škálovania upresnenými pri výsledkoch.

	Počet	CPU jadrá	RAM	Disk
Worker node	3	8	16GB	200GB
Kafka broker	3	12	96GB	8TB

Tabuľka 4.2: Zdroje pre testovacie meranie



Obr. 4.1: Diagram infraštruktúry pre merania

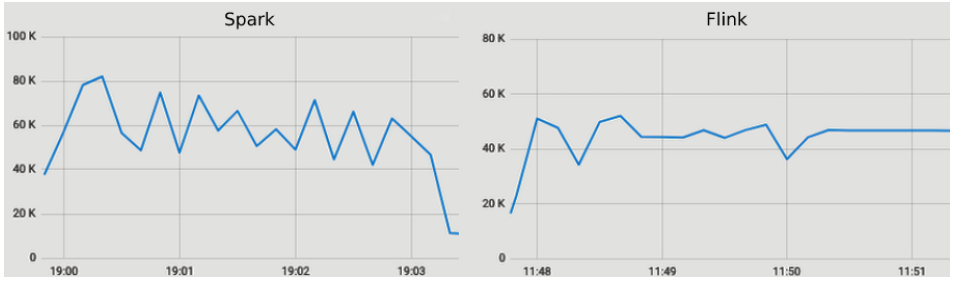
4.2.3 Výsledky merania

Pri prvom meraní som program spúšťal na troch uzloch, z čoho dva boli výpočtové uzly – Spark Worker alebo Flink TaskManager a jeden riadiaci uzol – Spark Driver alebo Flink JobManager. Každému výpočtovému uzlu som alokoval jedno jadro a 12GB pamäte, riadiacemu uzlu jedno jadro a 4GB pamäte.

Na grafe môžeme vidieť veľký rozdiel v latencií, Spark dosahuje priemerné latencie okolo 8 sekúnd, narozdiel od Flink-u, kde môžeme vidieť hodnoty okolo 300 milisekúnd. Čo sa týka stability, môžeme pozorovať, že Flink je pri použití týchto zdrojov náchylnejší na skoky v latenciách, kde Spark si udržuje stabilné hodnoty.

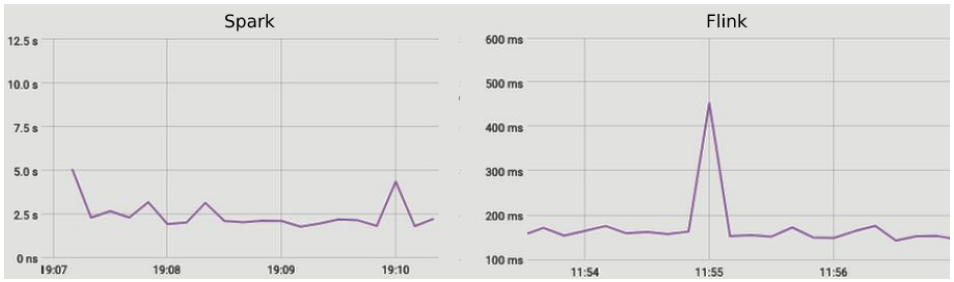
Obr. 4.2: Prvé meranie – latencie

Rozdiely v nameranej priepustnosti môžu byť spôsobené vyšším zaťažením hardvéru a tým znížením počtu generovaných správ. Ak by program nestíhal spracovávať správy, bolo by to viditeľné aj na latenciách, preto tento rozdiel považujem za zanedbateľný.



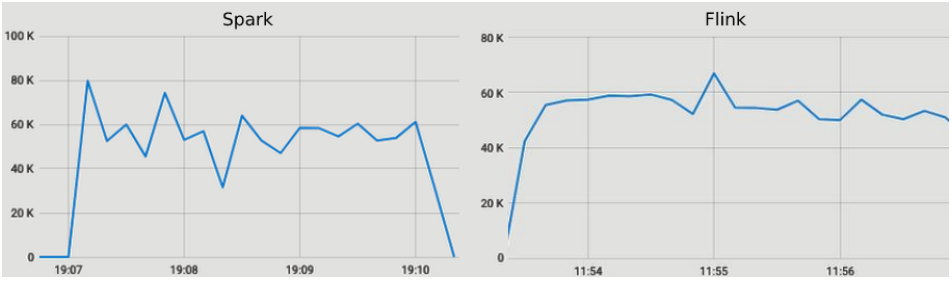
Obr. 4.3: Prvé meranie – priepustnosť

Pri druhom meraní som použil podobnú konfiguráciu, ale zvýšil som počet virtuálnych jadier na 5 pre každý výpočtový uzol. Tu môžeme vidieť značné zlepšenie v latenciách Spark-u. Pri Flink-u sú latencie podobné ako pri prvom meraní, ale zaznamenal som stabilnejšie hodnoty.



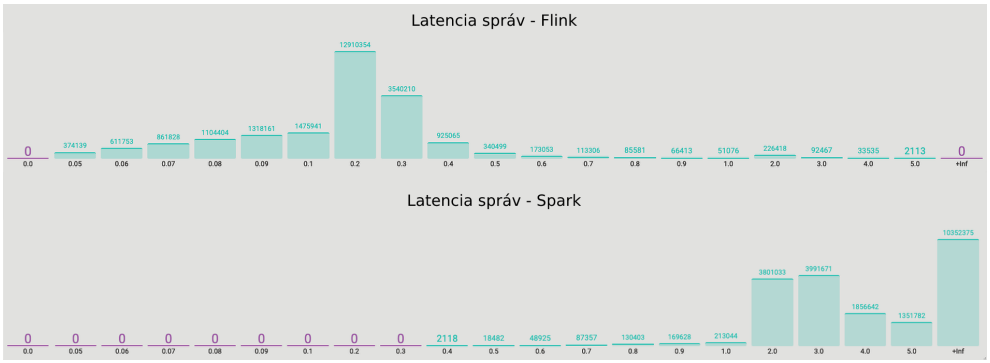
Obr. 4.4: Druhé meranie – latencie

Pri meraniach sa mi nepodarilo naraziť na limit priepustnosti v prípade oboch frameworkov. Niektoré výskumy však ukazujú, že Spark by mal zvládať vyššiu priepustnosť ako Flink pri použití rovnakých zdrojov [35].



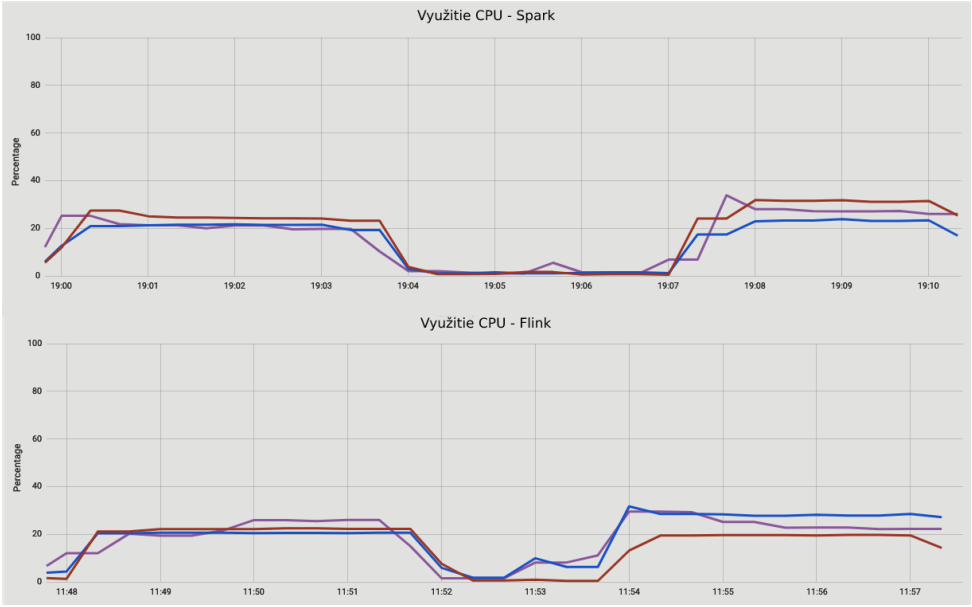
Obr. 4.5: Druhé meranie – priepustnosť

Pri meraní som sa stretol s problémom so synchronizáciou času medzi jednotlivými strojmi, ktorý môže čiastočne výsledky skresľovať. Pri veľkých rozdieloch sa môže dokonca stať, že sa objavia záporné latencie, čo v skutočnosti samozrejme nieje možné. Problém som riešil synchronizáciou času s NTP serverom (Network Time Protocol), s ktorým stroje synchronizujú svoj čas. Tento prístup stále nezaručuje úplnú presnosť, nameral som rozdiely rádovo v desiatkach milisekúnd, ale podarilo sa mi eliminovať záporné hodnoty. Táto citlivosť pre porovnanie týchto dvoch frameworkov postačuje, keďže je rozdiel latencií medzi frameworkami značný. Nižšie je vykreslený súhrnný histogram odozvy prijatých správ z oboch meraní.

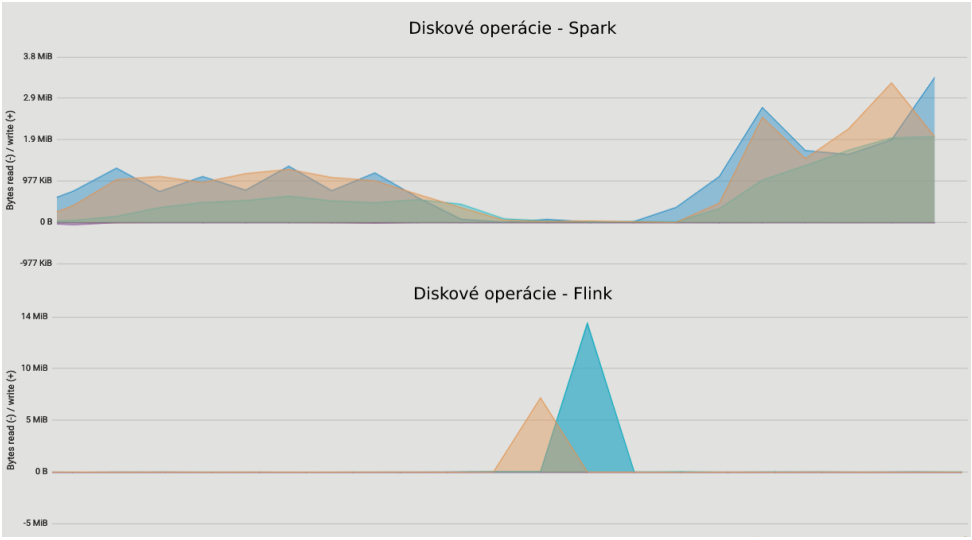


Obr. 4.6: Histogram latencií prijatých správ v sekundách

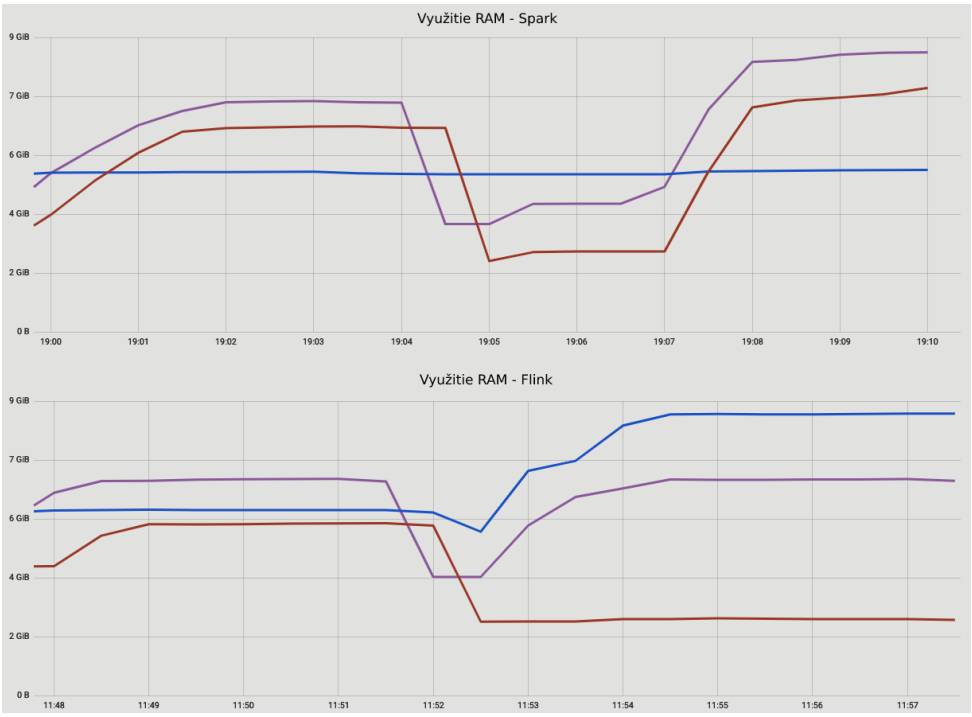
Nasledujúce grafy zobrazujú využitie hardvéru počas oboch meraní. Medzi meraniami je krátka prestávka, kde možno pozorovať stav, kedy aplikácia nebeží. Pri porovnávaní som nezaznamenal žiadne markantné rozdiely v žiadnom ohľade, okrem využitia disku, kde je Flink konzervatívnejší, čo môže byť spôsobené aj základnou konfiguráciou



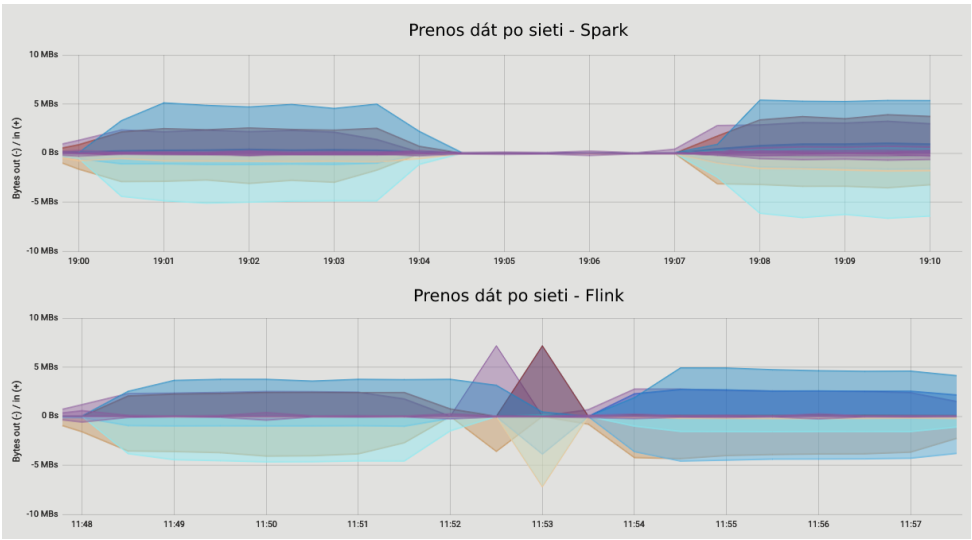
Obr. 4.7: Využitie CPU v percentách



Obr. 4.8: Diskové operácie, čítanie (+) / zápis (–)



Obr. 4.9: Využitie RAM



Obr. 4.10: Prenos dát po sieti, prijaté (+) / odoslané (–)

4.3 Voľba a argumentácia

Na základe porovnania vlastností a výsledkov meraní som pre architektúru zvolil Apache Flink. Podporuje všetky najmodernejšie funkcie pre streamovú analýzu dát, podobne ako aj Spark Structured Streaming. Pri implementácií mi prišlo rozhranie Flinku o trochu sympatickejšie a intuitívnejšie, rozhranie Sparku je však tiež na vysokej úrovni. Najväčšie rozdiely je možné badať vo výkone, kde Flink zažiaril svojimi nízkymi latenciami. Spark by mal zvládať vyššiu priepustnosť ako Flink pri rovnakých zdrojoch [35], avšak latencie sú niekoľkonásobne vyššie. Voľba preto dosť závisí od prípadu užitia. Prípadné požiadavky na vyššiu priepustnosť vieme vyriešit pri Flinku vyšším škálovaním, pri Sparku sa však k nižším latenciám nedopracujeme žiadnym spôsobom.

Výhodou Flinku je tiež integrácia s monitorovacím systémom Prometheus a o niečo jednoduchšie ladenie výkonu. Zaznamenal som tiež vysoké tempo vývoja tohoto frameworku, čo môže byť veľké plus do budúcnosti, kde funkcie ako napríklad automatické škálovanie je už momentálne vo fáze implementácie.

Architektúra vďaka svojej modularite a flexibilite umožňuje použitie oboch frameworkov zároveň. Ak by boli požiadavky napríklad na jednoduchú migráciu batchových programov implementovaných v Sparku, je možné využiť aj tento framework spolu s Flinkom na spoločnom clustri. Pri vytváraní prototypu sa však budem sústrediť na Apache Flink.

Kapitola 5

Prototyp

Pre overenie a demonštráciu funkčnosti navrhnutého prostredia som vytvoril prototyp, ktorý používa a prepája zvolené technológie. Ako základ pre prototyp som sa rozhodol použiť Docker, z dôvodu jednoduchej prenosnosti a reprodukcie môjho výsledku. Každá komponenta z navrhnutej architektúry beží ako jednouzlová aplikácia v samostatnom kontajneri. Pre orchestráciu a sieťové prepojenie jednotlivých kontajnerov som použil Docker Compose, celý prototyp je potom možné automaticky zostaviť pomocou jedného príkazu.

Prototyp by bolo možné rozšíriť pridaním viacerých uzlov do Hadoop alebo Kafka clustra pre experimenty so správaním pri výpadku jedného z nich. Garanciu funkčnosti pri výpadku nám ale spomenuté technológie poskytujú, preto sa v prototype skôr zameriavam na overenie spolupráce všetkých technológií a všeobecnú funkčnosť tohto návrhu.

5.1 Tvorba prototypu

Pri tvorbe prototypu som sa rozhodol použiť niektoré existujúce Docker obrazy z oficiálneho repozitára Docker Hub:

- Apache Kafka: open-source obraz *wurstmeister/kafka* vytvorený užívateľom wurstmeister, distribuovaný pod licenciou Apache License 2.0, dostupný na: `https://hub.docker.com/r/wurstmeister/kafka`
- Apache ZooKeeper: open-source obraz *wurstmeister/zookeeper* vytvorený užívateľom wurstmeister, distribuovaný pod licenciou Apache License 2.0, dostupný na:
  `https://hub.docker.com/r/wurstmeister/zookeeper`

- Prometheus: open-source obraz *prom/prometheus* priamo od tvorcov tejto technológie distribuovaný pod licenciou Apache License 2.0, dostupný na: `https://hub.docker.com/r/prom/prometheus`
- Grafana: open-source obraz *grafana/grafana* priamo od tvorcov tejto technológie distribuovaný pod licenciou GNU Affero General Public License v3.0 , dostupný na: `https://hub.docker.com/r/grafana/grafana`

Obrazy pre Hadoop cluster a Flink som sa rozhodol vytvoriť sám pre demonštráciu toho, ako by prebiehala manuálna inštalácia na reálne stroje:

- *hadoop*: jednouzlový Hadoop cluster (HDFS, YARN)
- *launcher*: kontajner obsahujúci Flink inštaláciu, aplikáciu a jej konfiguráciu, umožňujúci spúšťanie aplikácií na Hadoop clustri

Pri tvorbe obrazu *hadoop* som využil návod pre inštaláciu jednouzlovej inštancie Hadoop-u, ktorý sa nachádza v oficiálnej dokumentácií [34]. Pre automatizáciu tohto procesu som vytvoril Dockerfile, ktorý obsahuje jednotlivé kroky tohto postupu a tým automatizuje vytvorenie kontajnera. Obraz je založený na existujúcom obraze *openjdk:8-jdk-buster*, ktorý predstavuje operačný systém Debian Buster s predinštalovanými balíčkami Java, ktorú Hadoop vyžaduje pre svoj beh.

Dockerfile sa stará o inštaláciu závislostí, stiahnutie archívu, ktorý obsahuje Hadoop balíčky a kopírovanie konfigurácie zo stroja hostiteľa. Pri spustení kontajnera sa spustí entrypoint skript, ktorý pripraví potrebné premenné prostredia a spustí YARN a HDFS.

```
FROM openjdk:8-jdk-buster

RUN apt-get update -y
RUN apt-get install vim wget ssh -y

ENV HADOOP_HOME /opt/hadoop-3.3.0
ENV JAVA_HOME /usr/local/openjdk-8

WORKDIR /opt/

RUN wget "https://downloads.apache.org/hadoop
/common/hadoop-3.3.0/hadoop-3.3.0.tar.gz"

RUN tar -zxf hadoop-3.3.0.tar.gz && rm hadoop-3.3.0.tar.gz

RUN PATH=$PATH:$HADOOP_HOME/bin/

RUN ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
RUN cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
RUN chmod 0600 ~/.ssh/authorized_keys

COPY ./conf/* $HADOOP_HOME/etc/hadoop/
RUN cat $HADOOP_HOME/etc/hadoop/hadoop-env >>
    $HADOOP_HOME/etc/hadoop/hadoop-env.sh

ENTRYPOINT $HADOOP_HOME/etc/hadoop/entrypoint.sh
```

Výpis kódu 5.1: Dockerfile pre hadoop kontajner

Dockerfile pre obraz *launcher* vyzerá veľmi podobne, pretože taktiež potrebuje mať nainštalovaný Hadoop pre komunikáciu s YARN a HDFS, ktorý beží v kontajneri *hadoop*. Po správnej konfigurácií je možné prácu so súbormi na HDFS robiť priamo z tohto kontajnera bez nutnosti pripájania sa cez kontajner *hadoop*, rovnako je možné požadovať zdroje od YARN-u, čo využíva Flink pri spúšťaní aplikácií. Navyše obsahuje inštaláciu Flink balíčkov a konfigurácie.

5.2 Konfigurácia a spustenie

Štruktúru súborov v prototype som vytvoril na základe príslušnosti k jednotlivým kontajnerom. V koreňovom adresári prototypu sa nachádzajú zložky pomenované podľa názvu kontajneru, ktoré obsahujú príslušné konfiguračné súbory alebo iné užitočné skripty. V tejto sekcii v jednoduchosti predstavím minimálnu konfiguráciu, ktorá je vyžadovaná pre vzájomnú komunikáciu kontajnerov a správne fungovanie.

Základ prostredia tvorí Hadoop, preto začnem konfiguráciou tejto technológie. Pri konfigurácií YARN je potrebné nastaviť minimálne nasledovné parametre v súbore `yarn-site.xml` z dôvodu pripojenia na *ResuorceManager*.

```
<property>
    <name>yarn.resourcemanager.address</name>
    <value>hadoop:8032</value>
</property>

<property>
    <name>yarn.nodemanager.address</name>
    <value>hadoop:0</value>
</property>
```

Výpis kódu 5.2: Konfigurácia yarn-site.xml

Vďaka tomu, že som použil Docker Compose, jednotlivé kontajnery patria do jednej siete a vedia vzájomne komunikovať. Taktiež je možné používať miesto IP adries názov kontajnera pre jednoduchosť. Obdobne je pre HDFS potrebné nastaviť v súbore `core-site.xml` adresu a port cez ktoré je možné pripojenie na súborový systém. Rovnaká konfigurácia je použitá aj v kontajneri *launcher*. Kontajner *hadoop* ešte navyše obsahuje súbor `hdfs-site.xml`, kde sa nachádza nastavenie replikácie. V tomto súbore som taktiež pre jednoduchosť nastavil, aby bolo možné HDFS používať bez použitia prístupových práv.

Po tejto minimálnej konfigurácií je možné spúšťať programy z kontajnera *launcher*, tak aby bežali priamo na Hadoop clustri.

Pre Flink v prototype postačí základná konfigurácia, kde som doplnil konfiguráciu pre Prometheus. Vďaka tomuto Flink poskytuje HTTP rozhranie na uvedených portoch, kde vystavuje svoje metriky, ktoré môže Prometheus zbierať a uchovávať.

```
metrics.reporters: prom
metrics.reporter.prom.class:
    org.apache.flink.metrics.prometheus.PrometheusReporter
metrics.reporter.prom.port: 9990-9999
```

Výpis kódu 5.3: Časť konfigurácie Apache Flink

Konfigurácia Promethea potom spočíva v nastavení adries a portov, z ktorých má metriky zbierať, prípadne ďalšie doplnkové nastavenia ako napríklad interval zberu a podobne. Taktiež je možné definovať alerty v súbore `alerts.yml`. Pre vizualizáciu dát je v Grafane potrebné pridať ako zdroj dát Prometheus, kde stačí vyplniť adresu tohto servera, v tomto prípade `prometheus:9090`.

Konfigurácia Kafka a Zookeper kontajneru je definovaná pomocou premenných prostredia, tieto hodnoty sú špecifikované priamo v `docker-compose.yml`. Vďaka konfigurácií, ktorú som vytvoril je možné do Apache Kafka zapisovať z vnútornej siete (iných kontajnerov v spoločnom súbore `docker-compose.yml`) na adrese `kafka:9093`, zároveň aj z hostiteľského operačného systému na adrese `localhost:9092`. Testovacie skripty pre zápis preto môžeme spúšťať aj mimo tohto prostredia.

Pre spustenie prototypu je potrebné mať správne nainštalovaný Docker a Docker Compose. Po tomto kroku je zostavenie a spustenie veľmi jednoduché, stačí v zložke prototyp zavolať príkaz `docker-compose up`, prípadne pridať prepínač `-d` pre spustenie na pozadí, následne začne Docker zostavovať kontajnery. Tento proces môže zabrať niekoľko minút z dôvodu sťahovania balíčkov z internetu a inštalácie. Po prvom zostavení sa kontajnery automaticky spustia, prehľad aktuálne bežiacich kontajnerov môžeme vidieť po spustení príkazu `docker ps`.

Po spustení a inicializácií kontajnera *hadoop* je možné spúšťať Flink aplikácie z kontajnera *launcher*. Pre pripojenie na príkazový riadok slúži príkaz `docker exec -it <kontajner> bash`.

5.3 Overenie funkčnosti a testovanie

Pre demonštráciu funkčnosti som použil aplikáciu na detekciu anomálií, ktorú som popisoval v predošlej kapitole s drobným rozšírením. Na spustenie tejto aplikácie som vytvoril jednoduchý skript, ktorý sa nachádza v kontajneri *launcher* na umiestnení `/www/scripts/run_job.sh`.

5.3.1 Distribúcia a škálovanie

Distribúcia v zmysle presunu spustiteľného kódu a konfigurácie je riešená pomocou Docker kontajnera *launcher*. Tento kontajner tak nahrádza tradičné aplikačné balíky (napríklad .deb). Spustenie kontajneru je možné na ľubovolnom clustri (aj reálnom) po vhodnej konfigurácií.

O distribúciu v zmysle spúšťania výpočtov na viacerých strojoch sa v tomto prípade stará YARN. V prípade tohto prototypu, kde som použil jednouzlovú variantu Hadoop-u sa aplikácia distribuuje iba na jeden uzol. Pridaním ďalších YARN NodeManager by bolo možné jednoducho distribuovať výpočty na viac uzlov, podľa nastavení škálovania. Ako som spomínal už v úvode tejto kapitoly, primárnym účelom tohto prototypu bolo otestovať kompatibilitu jednotlivých technológií a nie vlastnosti, ktoré nám už tieto technológie garantujú.

Škálovať aplikácie je možné pomocou úpravy konfigurácie, prípadne pomocou prepínačov pri spustení. Nižšie uvádzam príkaz, ktorý spúšťa aplikáciu a obsahuje špecifikované zdroje, o ktoré sa žiada pri spustení. Pri zvýšení nárokov je možné konfiguráciu upraviť a aplikácií prideliť viac zdrojov.

```
flink run -t yarn-per-job \
       -Dtaskmanager.numberOfTaskSlots=1 \
       -Dparallelism.default=1 \
       -Dtaskmanager.memory.process.size=4g \
       -Dtaskmanager.memory.jvm-overhead.max=1g \
       -Djobmanager.memory.process.size=2g \
       /www/job/flink-testjob-0.1-all.jar /www/job/application.conf
```

Výpis kódu 5.4: Ukážka parametrov pre škálovanie Flink aplikácie

Vývojári Flink-u aktuálne pracujú aj na možnosti automatického škálovania počas behu aplikácie, čo je do budúcnosti veľká výhoda, hlavne pri použití cloudových výpočtových clustrov, kde sa cena prenájmu odvíja od spotrebovaných zdrojov. Škálovanie celej architektúry je možné dosiahnuť pridaním ďalších kontajnerov do Hadoop clustra, rovnako aj Kafka alebo ZooKeeper clustra.

5.3.2 Monitoring a alerting

Aby som overil funkčnosť monitoringu, teda prepojenia Flink-u a Promethea, vytvoril som v testovacej aplikácií akumulátor (counter), ktorý meria počet vzniknutých anomálií v dátach.

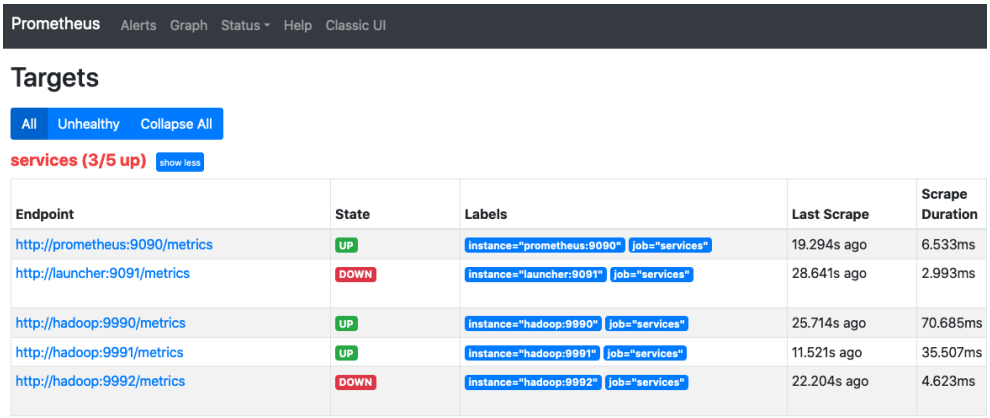
```
@transient private var outlierCount: Counter = _

...

if (flag == "outlier") outlierCount.inc()
```

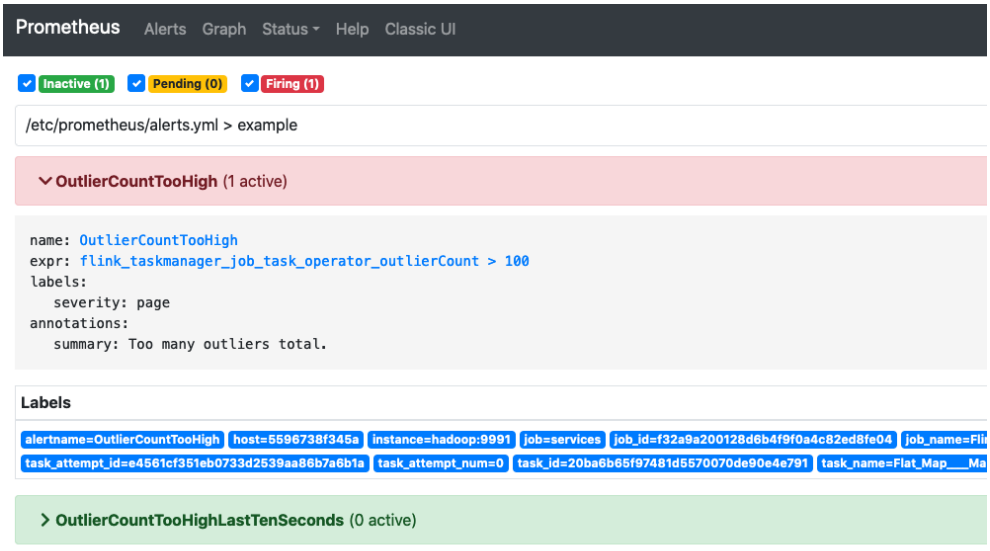
Výpis kódu 5.5: Ukážka Flink akumulátora

Tento akumulátor môže byť obnovený zo stavu pri zastavení aplikácie s vytvorením savepoint-u alebo z checkpoint-u pri neočakávanom výpadku. Po spustení aplikácie je táto metrika spolu s inými vystavená pre Prometheus na nastavenej adrese. Vo webovom rozhraní Promethea je možné skontrolovať, ktoré rozhrania sú aktívne:



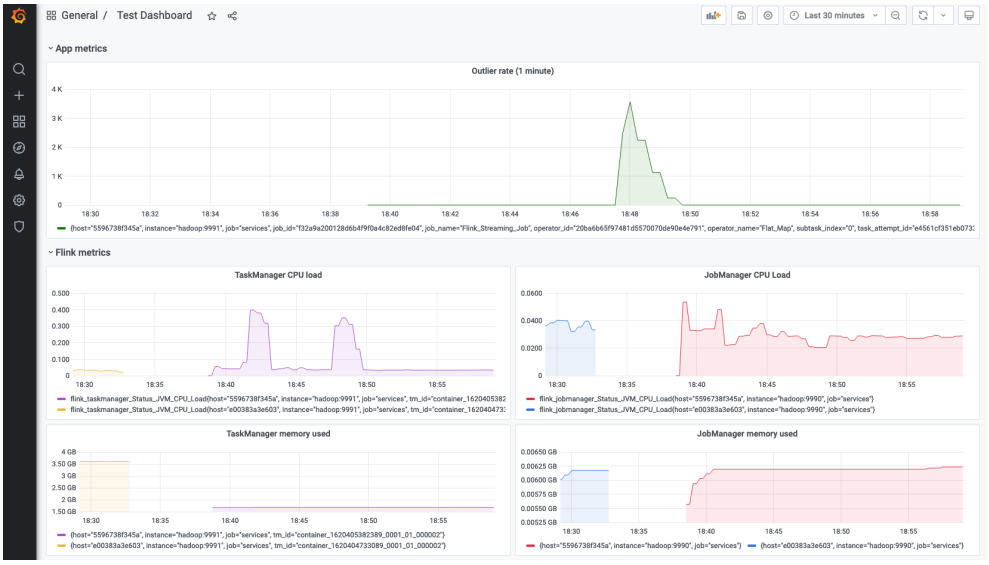
Obr. 5.1: Web rozhranie systému Prometheus

V tomto rozhraní je možné vytvárať a testovať dotazy v jazyku PromQL a grafovať získané výsledky. Na základe týchto dotazov je možné vytvoriť pravidlá pre alerting. Pre testovacie účely som vytvoril pravidlo, ktoré kontroluje, či je celkový počet anomálií nižší ako 100. V záložke alerts je potom možné sledovať stav týchto pravidiel. Prometheus alerting je možné ešte rozšíriť doplnkom zvaným Alertmanager, ktorý umožňuje výstrahy posielať napríklad emailom alebo iným spôsobom.



Obr. 5.2: Pravidlá pre alerting

Z Promethea je potom možné dáta vizualizovať v Grafane. Pre ukážku som vytvoril panel (v Grafane nazývaný ako *dashboard*), ktorý vizualizuje mnou vytvorenú metriku spolu s ďalšími základnými metrikami Flink aplikácie.



Obr. 5.3: Grafana dashboard

5.3.3 Zotavenie z výpadku, aktualizácie

Keďže agregačná funkcia, ktorú som implementoval pridáva na výstup značku pri nedostatočnom počte dát pre daný kľúč, je možné jednoducho skúsiť obnovenie zo stavu pri vypnutí alebo páde programu.

Pri teste výpadku som bežiacu aplikáciu zastavil cez YARN pomocou príkazu `yarn application -kill <app_id>`. To zabezpečí že YARN násilne odoberie zdroje aplikácii, ktorá nemá čas zareagovať na ukončenie. Takto môžno simulovať neočakávaný pád hardvéru.

Pri opätovnom spustení som k príkazu pre spustenie pridal argument `-s` s cestou k poslednému checkpointu. Checkpoint obvykle používa Flink vnútorne pre automatické obnovy pri páde jedného z TaskManagerov, je ale možné použitie aj v takomto prípade. Pomocou consumer skriptu som skontroloval, že vo výstupných dátach sa nenachádzajú žiadne značky indikujúce nedostatočný počet dát, čo znamená, že stav bol úspešne obnovený a aplikácia pokračuje vo výpočtoch zo stavu posledného checkpointu. V checkpointe sa taktiež nachádza aj pozícia vo vstupnom zdroji dát, nestane sa teda, že by niektoré správy boli započítané do agregácie viackrát.

Pri teste vypnutia, ktoré sa používa napríklad pri aktualizácií zdrojového kódu aplikácie som použil bash príkaz pre vytvorenie savepoint-u spustený v launcher kontajneri:

```
flink savepoint <flink_job_id> hdfs:///path -yid <yarn_app_id>
```

Pri opätovnom štarte som znova pridal ako argument k príkazu spustenia cestu, kde je savepoint uložený. Z neho si aplikácia prečíta potrebné informácie pre obnovenie stavu a pozície. Podmienkou je, aby štruktúra stavu bola rovnaká ako v predchádzajúcej verzii. V prípade, že by aktualizovaný kód obsahoval zmenu štruktúry, je potrebné implementovať funkciu ktorá dokáže starý stav zmigrovať na novú verziu.

Pre kontrolu dodržania *exactly-once* garancie na výstupe pri výpadku je potrebné používať konzumenta, ktorý dokáže pracovať s transakciami v Apache Kafka. Na tento účel už pôvodný skript z meraní nepostačuje, pri obnovení som spozoroval opätovný zápis dát, ktoré boli za poslednou pozíciou v čase vytvorenia checkpointu. To mi umožnilo overiť, že novo vypočítané dáta sa zhodujú s tými pred checkpointom, stav teda nebol nijak ovplyvnený výpadkom. Pri použití pokročilejšieho konzumenta (napr. Apache Flink Kafka Consumer) sú dáta z topicu prečítané až po dokončení transakcie, čo zaručí celkovú *exactly-once* garanciu. Túto vlastnosť som už ďalej neoveroval, pretože to Apache Flink garantuje [36]. Problém som zmieňoval už v sekcií 3.6, kde som uviedol, že nie všetky technológie vedia spolupracovať s Apache Kafka pri dodržaní *exactly-once*, preto je vhodné v prípade potreby zvoliť vhodnú alternatívu.

Ak počas doby, kedy je aplikácia vypnutá, vzniknú nové správy vo vstupnom topicu, je možné jednoducho overiť, že po opätovnom spustení sú tieto správy spracované a nestratia sa. To samozrejme platí za predpokladu, že nastavená dĺžka retencie je vyššia ako doba, po ktorú bola aplikácia nefunkčná.

5.4 Zoznam webových rozhraní

Webové rozhrania prístupné z hostiteľského operačného systému:

YARN `localhost:8088`

HDFS `localhost:9870`

Grafana `localhost:3000`, prihlasovacie meno: admin, heslo: admin

Prometheus `localhost:9090`

Flink `localhost:8088/proxy/<yarn_app_id>`

5.5 Príprava pre produkčné nasadenie

Na prototype som overil, že zvolené technológie spolu vedia komunikovať a spolupracovať a prostredie spĺňa špecifikované požiadavky. Tento prototyp však nieje vhodný pre produkčné používanie, keďže je určený iba pre použitie na jednom počítači, nie pre reálne distribuované používanie. Niektoré jeho časti je ale možné po vhodnej úprave používať.

Ak by sme chceli Flink aplikáciu spúšťať na reálnom clustri, je možné použiť vytvorený Docker kontajner, pretože po vhodnej úprave konfigurácie dokáže komunikovať s ľubovoľným Hadoop clustrom. Je potrebné aby sa verzia Hadoop-u v kontajneri zhodovala s verziou na clustri. V praxi sa ale často používajú virtualizované servery, kde je problematická ďalšia vrstva virtualizácie, preto by bolo vhodné použiť pre spúšťanie napríklad technológiu Kubernetes.

Pre správu väčšieho Hadoop clustra by bolo vhodné pridať ďalšie technológie, ktoré zjednodušujú konfiguráciu a správu strojov – napríklad Ambari (správa Hadoop súčastí) a Ansible (automatizovaná správa konfigurácií a údržba). Ďalej by bolo potrebné vyriešiť zabezpečenie týchto zariadení a nastavenie užívateľských a prístupových práv, ktoré sú v prototype pre jednoduchosť vypnuté.

Na výdaj dát, ktoré sú výsledkom spracovania by bolo vhodné zvoliť inú alternatívu pre výstupnú databázu. Ako som sa presvedčil pri testovaní, Apache Kafka je vhodná hlavne na presun dát medzi analytickými aplikáciam, nie na finálne ukladanie výsledkov, pretože dosiahnuť *exactly-once* garanciu na výstupe môže byť problematické.

Záver

V práci som predstavil úvod do spracovania veľých dát, prístupy k ich spracovaniu a rozdiely medzi nimi. Popísal som základy, na ktorých stavia streamový prístup a problémy, ktoré je potrebné riešiť pre vytvorenie spoľahlivého systému. Zoznámil som sa s technológiami, ktoré nám pomáhajú tieto problémy elegantne riešiť, predstavil som ich základné myšlienky a vlastnosti.

Pre vytvorenie funkčnej architektúry je potrebné tieto technológie vhodne prepojiť, tomu som sa venoval v úvode praktickej časti. Navrhol som prostredie, v ktorom je možné spúšťať analytické programy, ktoré spoľahlivo spracúvajú neohraničené datasety, sú jednoducho škálovateľné a poskytujú presné výsledky. V súčastnej dobe existuje viacero frameworkov, ktoré uľahčujú implementáciu týchto aplikácií, preto som sa venoval ich porovnaniu a meraniu výkonu, na základe čoho som zvolil Apache Flink pre použitie v prototype.

Na základe navrhnutej architektúry a zvoleného frameworku som vytvoril prototyp, ktorý toto prostredie dokáže simulovať na osobnom počítači (nevyžaduje cluster) a umožňuje overenie splnenia požadovaných vlastnosti. Prototyp je založený na technológií Docker, preto umožňuje jednoduchú prenosnosť a má nižšie nároky na hardvér, narozdiel od virtuálnych strojov. Popísal som postup tvorby tohto prototypu, overil som jeho funkčnosť a vysvetlil som, ako ho používať. Na záver som diskutoval možné vylepšenia, ktoré by bolo potrebné vyriešiť, aby bolo možné použitie tohto prostredia v produkcií.

Literatúra

[1] Google Cloud. *What is big data?* [online]. 12.6.2020 [cit. 21.01.2021]. Dostupné z: https://cloud.google.com/what-is-big-data

[2] Big data LDN. *BIG DATA: THE 3 VS EXPLAINED* [online]. [cit. 21.1.2021]. Dostupné z: https://bigdataldn.com/intelligence/big-data-the-3-vs-explained/

[3] DORBAND, E. John, Josephine Palencia RAYTHEON a Udaya RANAWAKE *Commodity Computing Clusters at Goddard Space Flight Center* [online]. [cit. 11.5.2021]. Dostupné z: https://spacejournal.ohio.edu/pdf/Dorband.pdf

[4] KAMBHAMPATI, Sunitha. Explore best practices for Spark performance optimization. In: *IBM Developer Blog* [online]. 30.6.2020 [cit. 11.5.2021]. Dostupné z: https://bigdataldn.com/intelligence/big-data-the-3-vs-explained/

[5] KLEPPMANN, Martin. *Designing Data-Intensive Applications*. Sebastopol: O'Reilly Media, 2017. ISBN 978-1-449-37332-0.

[6] MAAS, Gerard a François GARILLOT. *Stream Processing with Apache Spark*. Sebastopol: O'Reilly Media, 2019. ISBN 978-1-491-94424-0.

[7] Komunita vývojárov Apache Flink. Windows. In: *Apache Flink 1.12 documentation* [online]. 18.3.2021 [cit. 28.3.2021]. Dostupné z: https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/windows.html

[8] HUESKE, Fabian. Savepoints: Turning Back Time. In: *Ververica Blog* [online]. 14.10.2016 [cit. 28.3.2021]. Dostupné z: https://www.ververica.com/blog/turning-back-time-savepoints

[9] Wikipedia. *Apache Hadoop* [online]. [cit. 28.3.2021]. Dostupné z: https://en.wikipedia.org/wiki/Apache_Hadoop

[10] The Apache Software Foundation. *HDFS Architecture Guide* [online]. 10.10.2020 [cit. 28.3.2021]. Dostupné z: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[11] WHITE, Tom. *Hadoop: The Definitive Guide*. Štvrtá edícia. Sebastopol: O'Reilly Media, 2015. ISBN 978-1-491-90163-2.

[12] SHAPIRA, Gwen, Todd PALINO, Rajini SIVARAM a Neha NARKHEDE. *Kafka: The Definitive Guide*. Druhá edícia. Sebastopol: O'Reilly Media, 2021. ISBN 978-1-492-04301-0.

[13] Dattell. *What is ZooKeeper & How Does it Support Kafka?* [online]. 2020 [cit. 10.4.2021]. Dostupné z: https://dattell.com/data-architecture-blog/what-is-zookeeper-how-does-it-support-kafka/

[14] The Apache Software Foundation. *Apache Storm* [online]. 2019 [cit. 8.5.2021]. Dostupné z: https://storm.apache.org/about/simple-api.html

[15] The Apache Software Foundation. *Trident Tutorial* [online]. 2019 [cit. 12.5.2021]. Dostupné z: http://storm.apache.org/releases/current/Trident-tutorial.html

[16] Cloudera. *Apache Slider* [online]. 2021 [cit. 12.5.2021]. Dostupné z: https://www.cloudera.com/products/open-source/apache-hadoop/apache-slider.html

[17] The Apache Software Foundation. Core concepts. In:*Apache Samza Documentation* [online]. 2021 [cit. 8.5.2021]. Dostupné z: https://samza.apache.org/learn/documentation/1.6.0/core-concepts/core-concepts.html

[18] ZAHARIA, Matei, Mosharaf CHOWDHURY, Michael J. FRANKLIN, Scott SHENKER a Ion STOICA *Spark: Cluster Computing with Working Sets* [online]. 2011 [cit. 8.5.2021]. Dostupné z: https://amplab.cs.berkeley.edu/wp-content/uploads/2011/06/Spark-Cluster-Computing-with-Working-Sets.pdf

[19] Wikipedia. *Apache Spark* [online]. 23.4.2021 [cit. 8.5.2021]. Dostupné z: https://en.wikipedia.org/wiki/Apache_Spark

[20] Komunita vývojárov Apache Spark. *RDD Programming Guide* [online]. 18.10.2020 [cit. 28.3.2021]. Dostupné z: https://spark.apache.org/docs/latest/rdd-programming-guide.html

[21] Komunita vývojárov Apache Spark. *Spark SQL, DataFrames and Datasets Guide* [online]. 16.2.2020 [cit. 28.3.2021]. Dostupné z: https://spark.apache.org/docs/latest/sql-programming-guide.html

[22] Komunita vývojárov Apache Spark. *Spark Streaming Programming Guide* [online]. 15.7.2020 [cit. 8.5.2021]. Dostupné z: https://spark.apache.org/docs/latest/streaming-programming-guide.html

[23] Komunita vývojárov Apache Spark. *Structured Streaming Programming Guide* [online]. 20.2.2021 [cit. 8.5.2021]. Dostupné z: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

[24] KHUDAIRI, Sally. The Apache Software Foundation Announces Apache™ Flink™ as a Top-Level Project. In: *The Apache Software Foundation Blog* [online]. 12.1.2015 [cit. 8.5.2021]. Dostupné z: https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces69

[25] The Apache Software Foundation. *What is Apache Flink? — Operations* [online]. [cit. 8.5.2021]. Dostupné z: https://flink.apache.org/flink-operations.html

[26] Komunita vývojárov technológie Prometheus. Overview. In: *Prometheus Docs* [online]. 13.12.2018 [cit. 18.4.2021]. Dostupné z: https://prometheus.io/docs/introduction/overview/

[27] Nested Software. *Calculating Standard Deviation on Streaming Data* [online]. 27.3.2018 [cit. 18.4.2021]. Dostupné z: https://nestedsoftware.com/2018/03/27/calculating-standard-deviation-on-streaming-data-253l.23919.html

[28] Grafana Labs. *Grafana* [online]. 2021 [cit. 11.5.2021]. Dostupné z: https://grafana.com/oss/grafana/

[29] Komunita vývojárov technológie Docker. Docker overview. In: *Docker Docs* [online]. 4.2.2020 [cit. 18.4.2021]. Dostupné z: https://docs.docker.com/get-started/overview/

[30] Weaveworks. *A Practical Guide to Choosing between Docker Containers and VMs* [online]. 16.1.2020 [cit. 18.4.2021]. Dostupné z: https://www.weave.works/blog/a-practical-guide-to-choosing-between-docker-containers-and-vms

[31] Komunita vývojárov technológie Docker. *Overview of Docker Compose* [online]. 23.3.2021 [cit. 18.4.2021]. Dostupné z: https://docs.docker.com/compose/

[32] NOWOJSKI, Piotr, Mike Winters. An Overview of End-to-End Exactly-Once Processing in Apache Flink (with Apache Kafka, too!). In: *Flink Blog* [online]. 1.3.2018 [cit. 25.4.2021]. Dostupné z: https://flink.apache.org/features/2018/03/01/end-to-end-exactly-once-apache-flink.html

[33] Wikipedia. *Standard score* [online]. 13.4.2021 [cit. 25.4.2021]. Dostupné z: https://en.wikipedia.org/wiki/Standard_score

[34] Komunita vývojárov technológie Hadoop. *Hadoop: Setting up a Single Node Cluster.* [online]. 3.1.2021 [cit. 28.3.2021]. Dostupné z: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html

[35] YAVUZ, Burak. Benchmarking Structured Streaming on Databricks Runtime Against State-of-the-Art Streaming Systems. In: *Databricks Blog* [online]. Databricks, 11.10.2017 [cit. 25.4.2021]. Dostupné z: https://databricks.com/blog/2017/10/11/benchmarking-structured-streaming-on-databricks-runtime-against-state-of-the-art-streaming-systems.html

[36] Komunita vývojárov Apache Flink. Apache Kafka Connector. In: *Apache Flink Documentation* [online]. 16.2.2021 [cit. 11.5.2021]. Dostupné z: https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/connectors/datastream/kafka/

Dodatok A

Zoznam použitých skratiek

CSV Comma-separated values

HDFS Hadoop Distributed File System

HTTP Hypertext transfer protocol

SQL Structured query language

TCP/IP Transmission Control Protocol/Internet Protocol

XML Extensible markup language

YARN Yet Another Resource Manager

Dodatok **B**

Obsah priloženého DVD

- `readme.txt` ..... stručný popis obsahu DVD
- `thesis` ..... text práce
  - `src` ..... zdrojový kód textu práce vo formáte LaTeX
  - `thesis.pdf` ..... text práce vo formáte PDF
- `measurements` ..... príloha k porovnaniu frameworkov z kapitoly 4
  - `applications` ..... zdrojový kód aplikácií použitých pre meranie
  - `graphs` ..... grafy z merania
  - `raw_data` ..... dáta z merania
  - `scripts` ..... skripty použité pre meranie
- `prototype` ..... zdrojový kód prototypu, konfiguračné súbory
  - `readme.txt` ..... krátky popis použitia prototypu
  - `docker-compose.yml`
  - `hadoop`
  - `launcher`
  - `prometheus`
  - `grafana`
  - `host_scripts`