**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE**
**FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-5382-92323

# VYUŽITIE APACHE SPARK NA DETEKCIU A OŠETRENIE OBCHODNE RELEVANTNÝCH UDALOSTÍ

## BAKALÁRSKA PRÁCA

**2020** **Stanislav Pekarovič**

**SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE**
**FAKULTA ELEKTROTECHNIKY A INFORMATIKY**

Evidenčné číslo: FEI-5382-92323

# VYUŽITIE APACHE SPARK NA DETEKCIU A OŠETRENIE OBCHODNE RELEVANTNÝCH UDALOSTÍ

## BAKALÁRSKA PRÁCA

| | |
|---|---|
| Študijný program: | aplikovaná informatika |
| Študijný odbor: | informatika |
| Školiace pracovisko: | Ústav informatiky a matematiky |
| Vedúci záverečnej práce: | Mgr. Tomáš Fabšič, PhD. |
| Konzultant: | Ľubomír Tokár |

**Bratislava 2020** **Stanislav Pekarovič**

# SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

| | |
|---|---|
| Študijný program: | aplikovaná informatika |
| Autor: | Stanislav Pekarovič |
| Bakalárska práca: | Využitie Apache Spark na detekciu a ošetrenie obchodne relevantných udalostí |
| Vedúci záverečnej práce: | Mgr. Tomáš Fabšič, PhD. |
| Konzultant: | Ľubomír Tokár |
| Miesto a rok predloženia práce: | Bratislava 2020 |

Cieľom práce bolo zoznámenie sa s technológiami Apache Kafka, Apache Spark a ich základnými konceptmi. V prípade Apache Spark bolo potrebné porovnať jednotlivé programovacie jazyky ako Java a Scala a taktiež bolo potrebné preveriť možnosti zaintegrovania kódu modelu v jazyku Python.

V práci sme do značnej miery popísali spracovanie dátových tokov a to tak ako z pohľadu Apache Kafka tak i Apache Spark. Pri Apache Spark sme do detailov popísali Apache Spark Streaming a taktiež aj Apache Spark Structured Streaming, ktorý sme využili aj počas implementácie.

V rámci práce boli identifikované dve obchodne prínosné udalosti, na ktoré má význam reagovať s krátkym oneskorením, a bolo implementované riešenie pre tieto problémy. Konkrétne sa jednalo o aplikáciu na detekciu fraudulentných platieb a aplikáciu, ktorá pomocou nástroja Drools vie filtrovať platby na základe biznis pravidiel.

Kľúčové slová: Apache Spark, Apache Kafka, Java, bankové systémy

# ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

| | |
|---|---|
| Study Programme: | applied Informatics |
| Author: | Stanislav Pekarovič |
| Bachelor's thesis: | Use Apache Spark to detect and treat business-relevant events |
| Supervisor: | Mgr. Tomáš Fabšič, PhD. |
| Consultant: | Ľubomír Tokár |
| Place and year of submission: | Bratislava 2020 |

The aim of the thesis was to get acquainted with technologies such as Apache Kafka, Apache Spark and their basic concepts. In the case of Apache Spark, it was necessary to compare individual programming languages such as Java and Scala, and it was also necessary to examine the possibilities of integrating the code of the model in Python.

In this thesis, we have largely described the processing of data streams, both from the perspective of Apache Kafka and Apache Spark. In the case of Apache Spark, we went into details of Apache Spark Streaming and also Apache Spark Structured Streaming, which we also used during the implementation part.

Within the thesis, two commercially beneficial events were identified to which it is important to respond without a delay and a solution was implemented for those issues. Specifically, it was an application for detecting fraudulent payments and an application that can use a tool called Drools to filter payments based on business rules.

Keywords: Apache Spark, Apache Kafka, Java, banking systems

# Poďakovanie

Ďakujem vedúcemu práce Ľubomírovi Tokárovi za konštruktívnu kritiku, pripomienky a rady, ktoré mi pomohli vytvoriť túto bakalársku prácu. Taktiež by som sa chcel poďakovať doc. Ing. Milanovi Vojvodi, PhD. za rady a pripomienky k formálnej stránke mojej práce. V neposlednom rade by som chcel vyjadriť svoju vďaku Slovenskej sporiteľni za to, že mi umožnili spolupracovať s nimi na tejto práci.

# Obsah

# Zoznam obrázkov a tabuliek

# Zoznam skratiek

| | |
|---|---|
| **API** | Application programming interface |
| **BRMS** | Business Rules Management System |
| **DAG** | Directed Acyclic Graph |
| **DRL** | Drools Rule Language |
| **DStream** | Discretized Stream |
| **HDFS** | Hadoop Distributed File System |
| **IT** | Information technology |
| **JSON** | JavaScript Object Notation |
| **JVM** | Java Virtual Machine |
| **KIE** | Knowledge Is Everything |
| **Pub/sub** | Publish/subscribe |
| **RAM** | Random Access Memory |
| **RDD** | Resilient Distributed Datasets |
| **SQL** | Structured Query Language |
| **WAL** | Write-ahead logging |

# Úvod

Každý informačný systém narába s dátami. Tieto dáta systémy produkujú a následne ukladajú na rôzne úložiská prípadne ich zobrazujú užívateľovi. Systémy dáta nielenže produkujú, ale taktiež dáta prijímajú či sa už jedná o vstupy používateľov alebo obnosy dát, ktoré vstupujú do systému z iných systémov. Dôležitým faktorom v otázke dát s ktorými systémy narábajú je nepochybne aj ich samotné množstvo. Môžu byť aplikácie, ktoré spracovávajú veľmi malý obnos údajov a taktiež rozhodne existujú i také, ktoré musia byť schopné narábať s obrovským množstvom dát.

Čo sa týka veľkosti dát, IDC predpovedá, že Global DataSphere[1] porastie z 33 Zettabytov na 175 Zettabytov do roku 2025. To je viac ako päť násobný nárast množstva dát, ktoré bude za potreby spracovať.[1] Napriek tomu je však toto množstvo opodstatnené nakoľko každým dňom pribúdajú množstvá zariadení či už v podobe serverov, osobných počítačov, smartfónov alebo inteligentných zariadení.

Bez ohľadu na množstvo dát, s ktorými aplikácie prichádzajú do styku postupom času, prišla na rad otázka toho, ako rýchlo vieme na prichádzajúce dáta alebo udalosti reagovať. Dôvodom, prečo je čas reakcie tak dôležitý je práve ten, že niektoré informácie, ktoré systémom prechádzajú môžu byť relevantné iba ak je na ne systém schopný zareagovať vo veľmi krátkom čase.

Napríklad, pre bezpečnostný systém, ktorý má na starosti teplotné senzory, môže byť užitočné pokiaľ vie užívateľa upozorniť takmer okamžite v prípade, že rapídne stúpla teplota v jednej z miestností. Na druhú stranu takáto informácia nebude mať pre používateľa veľký prínos pokiaľ bude upozornený na konci týždňa.

Avšak sú, samozrejme, aj aplikácie, ktoré nebenefitujú z krátkeho reakčného času na prichádzajúcu udalosť. Do takejto kategória môže patriť napríklad aplikácia slúžiaca na pravidelné sledovanie vlhkosti pôdy a vytvorenie mesačnej správy o stave v priebehu jednotlivých dní mesiaca.

V oboch prípadoch však zariadenia resp. systémy produkujú dátové toky, ktoré je potrebné spracovať a vyhodnotiť. Jedným zo zdrojom veľkého množstvá dát sú nepochybne aj bankové systémy. V nich vznikajú rôzne transakcie, procesné logy[2], záznamy či už z osobných návštev bánk či návštev webových stránok. Bez ohľadu na zdroj vzniku týchto dát však platí, že ak budú v dostatočne rýchlom čase správne interpretované môžu pomôcť

[1] Global DataSphere kvantifikuje a analyzuje množstvo údajov vytvorených, zachytených a replikovaných v ktoromkoľvek danom roku na celom svete.

[2] Log alebo aj žurnál je záznam nejakej činnosti, ktorú program ukladá ako informáciu o svojej činnosti.

napríklad:

- získať prehľad o aktuálnej činnosti klienta,
- vyriešiť potenciálne problémové situácie zákazníkov,
- v správny moment ponúknuť vhodnú službu klientovi,
- pomôcť pri detekcií fraudulentných transakcií,
- zvýšiť celkový výkon systému,
- zefektívniť analýzu spätnej väzby zákazníkov.

V bankovom sektore to môže tým pádom na jednu stranu priniesť kvalitnejšie služby avšak taktiež môže pomôcť chrániť existujúcich klientov danej banky.

Je teda zrejmé, že bankové systémy sa dennodenne musia vysporiadať s veľkým obnosom dát, čo si vyžaduje vhodné technológie, ktoré musia byť schopné spracovať tieto dáta. Bankový sektor sa však vyznačuje tým, že dáta, ktoré sa v ňom presúvajú sú spravidla citlivé a preto tieto systémy musia okrem efektívneho spracovania spĺňať aj rôzne ďalšie požiadavky ako napríklad bezpečnosť, odolnosť voči poruchám, zotaviteľnosť po poruchách a tak ďalej.

Práve z tohto dôvodu sme sa v spolupráci so Slovenskou sporiteľňou rozhodli preskúmať možnosti riešenia týchto problémov použitím technológií ako Apache Kafka a Apache Spark, ktoré popíšeme v tejto práci.

# 1 Apache Kafka

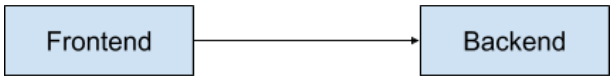
Streamované dáta alebo dátové toky predstavujú také dáta, ktoré sú generované nepretržite dátovými zdrojmi, ktoré zvyčajne tieto dáta zasielajú súčasne a v malých dávkach (rádovo kilobajtoch). Dátové toky by sa dali taktiež popísať ako sekvencia digitálne kódovaných signálov použitých na znázornenie informácií pri prenose. Takéto typy dát zahŕňajú širokú škálu údajov ako napríklad údaje generované mobilnými aplikáciami, informácie zo sociálnych sietí, informácie o bankových transakciách, telemetrické údaje z pripojených zariadení a tak ďalej.[2][3]

Apache Kafka je distribuovaná platforma pre prácu s takýmto typom dát. Táto platforma sa vyznačuje troma kľúčovými vlastnosťami[4]:

- funguje na princípe publikovania/odoberania správ alebo skrátene Pub/sub,
- ukladá dátové toky a zachováva ich odolné voči prípadným poruchám,
- spracováva dáta tak, ako sa vyskytnú.

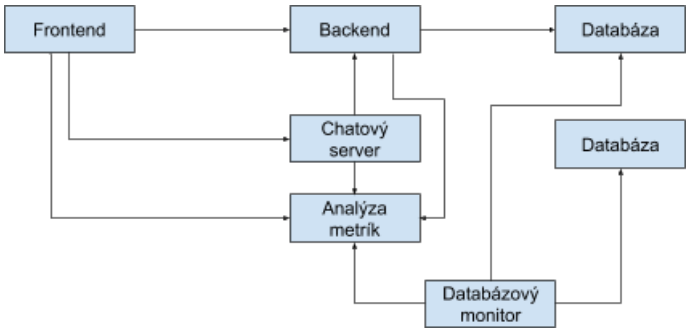
Pub/sub predstavuje vzor, ktorý je charakterizovaný odosielateľom (publisher), ktorý odosiela správu. V tomto prípade však táto správa nie je špecificky určená pre nejakého prijímateľa. Miesto toho aby odosielateľ v správe špecifikoval príjemcu túto správu iba klasifikuje. Príjemca alebo odberateľ (subscriber) sa následne rozhodne, ktoré správy bude odoberať.[5]

Takýto prístup nie je zvlášť potrebný v prípade, že sa jedná o jednoduchú aplikáciu. Napríklad systém pozostávajúci z klientskej časti (frontend), ktorý môže predstavovať webová stránka, a následne zo serverovej časti (backend) ako je možné vidieť na obr. 1. Takéto dva celky potrebujú nejako interagovať, a preto je potrebné vytvoriť spojenie medzi týmito dvoma celkami.




Obr. 1: Znázornenie komunikácie jednoduchej aplikácie.

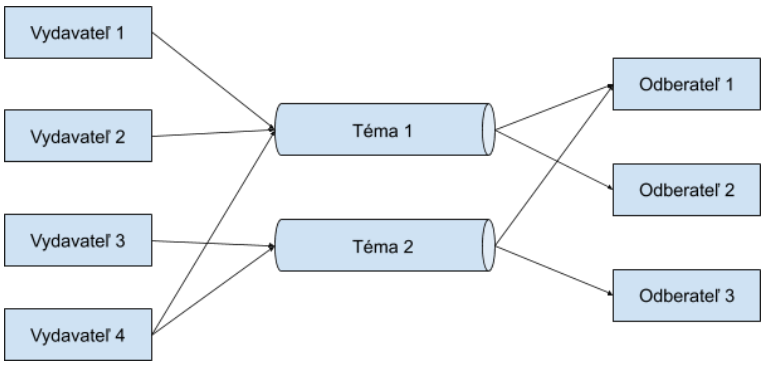
V takomto prípade nemusí byť použitie Apache Kafka potrebné. Avšak postupom času sa môže takýto systém začať rozvíjať, pridávať doň nové funkcie, databázy a tak ďalej. So zvyšujúcim sa počtom komponentov takejto architektúry sa však zvyšuje aj množstvo komunikačných kanálov, pomocou ktorých musia jednotlivé časti medzi sebou komunikovať. Preto takáto architektúra môže neskôr vyzerať nasledovne viď obr. 2.




Obr. 2: Znázornenie komunikácie pri rozšírení aplikácie.

Takáto architektúra sa s každým pridaným prvkom stáva viac a viac neprehľadnou, ťažšie sa v nej dá vyznať v zmysle, ktorý prvok komunikuje s ktorým. Taktiež sa pri pridaní nového prvku do systému musí takýto komponent prepojiť so všetkými ostatnými prvkami, s ktorými potrebuje komunikovať.

Práve tento problém je možné riešiť zakomponovaním Apache Kafka do systému. Pomocou Apache Kafka je možné eliminovať potrebu zostavovania separátneho spojenia medzi jednotlivými komponentami systému. Apache Kafka, ako sme už spomenuli, používa Pub/sub vzor a tým pádom jednotlivé komponenty systému budú pri použití Apache Kafka predstavovať vydavateľov a odberateľov.




Obr. 3: Schéma publikovania/odoberania správ.

Správy, ktoré produkujú vydavatelia sú kategorizované do tém (tieto koncepty bližšie popíšeme v nasledujúcich kapitolách). Téma (topic) slúži v tomto prípade ako komunikačný kanál. Vydavatelia tak produkujú svoje správy do tém a je na odberateľoch, z ktorých tém sa rozhodnú správy odoberať.

Tento prístup dokáže práve komplikovanejšej architektúre pomôcť. Nakoľko ak je potrebné pridať nový komponent do už existujúceho systému je tak možné spraviť bez uzatvárania individuálnych spojení medzi jednotlivými komponentami, s ktorými bude

komunikovať. Takýto komponent je možné evidovať ako odberateľa príslušnej témy, ktorej správy sú pre takýto element systému prínosné. Prípadne tento prvok môže predstavovať vydavateľa správ do už existujúcej témy.

## 1.1 Prípady použitia

V predošlej kapitole sme popísali ako môže Apache Kafka pomôcť v zjednodušení archituktúry väčších systémov. Avšak Apache Kafka má aj mnoho iných prípadov, kedy je možné ju využiť práve vďaka jej schopnosti narábať so streamovanými dátami.

**Zasielanie správ** Medzi tieto použitia patrí napríklad zasielanie správ. Kafka je využiteľná za predpokladu, že aplikácie potrebujú zasielať upozornenia svojim užívateľom. Systém, ktorý produkuje tieto správy sa tak nemusí zaujímať o to, ako bude daná správa odoslaná a taktiež akým štýlom bude formátovaná. Môže sa jednať napríklad o mail alebo SMS správu. Avšak túto časť môže mať na starosti iný systém, ktorý číta tieto správy a následne na ne reaguje. Tento prístup má tak výhodu v tom, že druhý systém môže mať na starosti formátovanie správ či zlúčenie sérií notifikácií do jednej a tak ďalej.

**Sledovanie aktivity** Ďalším z možných prípadov využitia Apache Kafka je sledovanie aktivity užívateľa. Takýto prístup môže byť užitočný napríklad pri spracovaní udalostí zasielaných napríklad z webových stránok. Udalosť v tomto prípade môžu predstavovať napríklad užívateľove kliknutia na jednotlivé linky na stránke alebo návštevy jednotlivých podstránok. Taktiež sa však môže jednať aj o komplexnejšie udalosti, ktoré môžu byť reprezentované napríklad pridaním príspevku, zmenu jeho statusu, profilovej fotky, dokúpením ponúknutej položky na stránke webového obchodu a tak ďalej. Motiváciou za takýmto prípadom použitia môže byť napríklad vytvorenie štatistiky pre ďalšiu biznis analýzu. Taktiež tieto údaje môžu byť použité napríklad na trénovanie modelov strojového učenia pre ponúknutie relevantnej ponuky zákazníkovi a tak ďalej.

**Logovanie** Mnoho informačných systémov produkuje veľa logov. Logy môžu obsahovať mnohokrát cenné informácie od záznamov toho, čo sa počas daných akcií dialo, až po chyby systému. Väčšinou platí, že čím väčší daný systém je, tým viac logov jeho jednotlivé komponenty produkujú. S použitím Apache Kafka môžu jednotlivé komponenty zasielať logy v podobe správ do jednej témy a následne môže byť monitorovací systém, ktorý bude konzumovať tieto správy, vyhodnocovať ich a následne môže upozorňovať na kritické problémy a tak ďalej. Takýto spôsob taktiež umožňuje jednoduché prepojenie so systémami na analýzu logov ako napríklad Eleasticsearch.[5]

## 1.2 Základné koncepty Apache Kafka

V predchádzajúcich kapitolách sme spomenuli pojmy, ktoré zavádza Apache Kafka, a to konkrétne pojmy ako: sprostredkovateľ (broker), vydavateľ (publisher), odberateľ (subscriber), téma (topic) a správy (message). V nasledujúcich kapitolách sa pokúsime tieto pojmy priblížiť a opísať do väčšieho detailu.

### 1.2.1 Pub/Sub a témy v Apache Kafka

Ako sme už spomínali, vydavateľ má za úlohu odoslať správu do príslušnej témy. Téma predstavuje neustále narastajúcu sekvenciu záznamov pričom každému záznamu, ktorý do tejto sekvencie pribudne je priradené jedinečné číslo - offset.

Z pohľadu vydavateľa nemá offset žiadnu závažnú zmenu - nemusí si offset odkladať ani s ním nijako ďalej narábať. Rozdiel však nastáva v prípade odberateľa. Odberateľ má na starosti odoberanie týchto správ z jednotlivých tém pričom on sám si určuje od akého offsetu začne čítať záznamy z témy. Je dôležité dodať, že po prečítaním správy odberateľom správa v Kafke nezanikne.

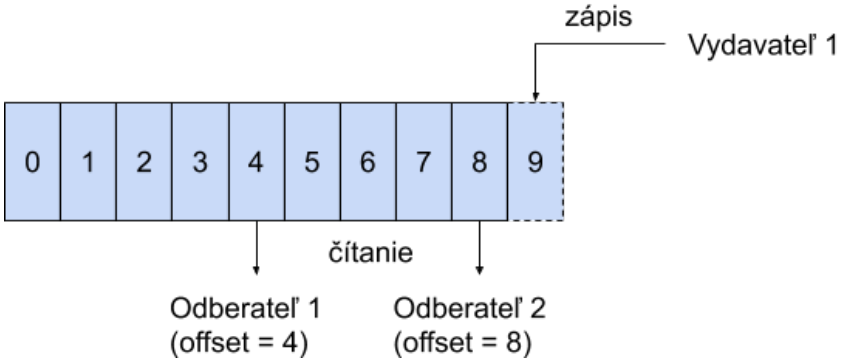
V prvej kapitole sme hovorili o tom, že Apache Kafka dokáže dáta taktiež ukladať. Zánik správ v Kafke je kontrolovaný retenciou, ktorá môže byť na každú tému nastavená zvlášť. Retencia v Kafke môže byť nastavená na určitý časový interval, po uplynutí ktorého nebude ďalej záznam v Kafke uchovávaný. Taktiež je možné ju nastaviť na určitú kapacitu, po ktorej presiahnutí budú správy zmazané.

Ako sme už spomenuli, odberateľ môže odoberať správy nejakej témy. Avšak Apache Kafka umožňuje užívateľovi definovať tzv. odberateľskú skupinu(consumer group). Odberateľská skupina tak umožňuje užívateľovi definovať skupinu odberateľov, ktorí môžu spoločne čerpať z tej istej témy. Za predpokladu, že sú daní odberatelia z tej istej skupiny, budú odoberať z tejto témy spoločne čiže každá zo správ bude prečítaná iba jeden krát - nastane prerozdelenie.

Ak by boli dvaja odberatelia, ktorí nie sú v tej istej skupine (avšak odoberajú z tej istej témy), nastane situácia, kedy každý z odberateľov prečíta každú správu z danej témy čiže v tomto prípade budú správy prečítané dvakrát.

V prípade odberateľov však zohráva svoju rolu aj offset. Offset tvorí jediné metadáta, ktoré sú ukladané v Kafke per-odberateľ. To dáva odberateľovi voľnosť v tom, ako bude správy z danej témy čítať. Taktiež to znamená, že každý z odberateľov môže čítať dáta z iného offsetu (nie sú na sebe závislí viď. obrázok 4).

Odberateľ môže v správach resp. offsetoch postupovať lineárne avšak taktiež môže postupovať v ľubovoľnom poradí. Taktiež je tým pádom možné jednotlivé správy znovu




Obr. 4: Vizualizácia správ s offsetmi v téme.

prehrať tým, že sa rozhodne zvoliť starší offset alebo môže preskočiť niektoré zo správ zvolením príslušného offsetu.

### 1.2.2 Partície

Správa v Kafke pozostáva z kľúču a hodnoty, ktoré tvoria pár. Taktiež je potrebné poznamenať, že kvôli efektívnemu spracovaniu týchto správ je téma rozdelená na partície. Samotný vydavateľ sa môže rozhodnúť, do ktorej partície zvolenej témy zašle správu a môže tak učiniť nasledovne:

- explicitne definuje číslo partície,
- definuje kľúč, na základe ktorého sa vypočíta hash partície (ak nie je zadané číslo partície).

Pokiaľ však vydavateľ nešpecifikuje číslo partície alebo nezadá kľúč, tak sa číslo partície zvolí Round Robinovým algoritmom.[6]

K zapisovaniu do partícií je taktiež potrebné dodať, že pokiaľ je záznam zapísaný do partície, tak nemôže byť zmenený. Čo sa týka poradia správ tak ich usporiadanosť je garantovaná len v rámci partície nie medzi partíciami.

Dôvod, prečo sa v Kafke delia témy na partície je predovšetkým škálovateľnosť. Tá sa dá docieliť v rámci témy tým, že z nej bude odoberať správy celá skupina odberateľov. Pokiaľ by v tejto skupine bol iba jeden odberateľ, ten by v takom prípade odoberal správy zo všetkých partícií.

Avšak ak by bol pridaný do tejto skupiny ďalší, tak by boli jednotlivé partície medzi nimi prerozdelené, čím by sa zvýšila miera paralelného spracovania. Ak by bolo v tejto skupine toľko odberateľov koľko je partícií odoberanej témy, tak by každému prislúchala práve jedna partícia. Z toho taktiež vyplýva, že nemá zmysel mať v skupine viac

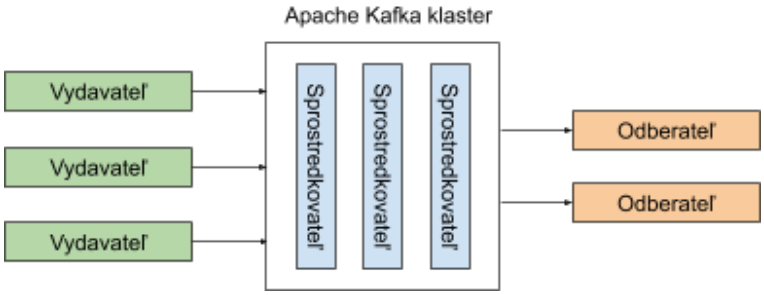
odberateľov ako je partícií v téme nakoľko zvyšní odberatelia budú nečinní.[3]

### 1.2.3 Sprostredkovatelia a Kafka klaster

Kafka klaster pozostáva vždy z jedného alebo viacerých serverov pričom každý z týchto serverov môže mať jeden alebo viacero serverových procesov. Práve takýto proces sa nazýva sprostredkovateľ (broker). Už spomínané témy sa vytvárajú práve v kontexte týchto procesov.[7]

Úlohou sprostredkovateľa je taktiež prijímať správy a priraďovať im príslušný offset a následne tieto správy odovzdávať do úložiska na disku. Ukladanie na disk je dôležitou úlohou sprostredkovateľa nakoľko diskovou retenciou správ sa umožní čítanie týchto správ aj po dlhej dobe odkedy boli prijaté.

Kafka je navrhnutá tak, že viacerí sprostredkovatelia formujú klaster. V ňom má jeden zo sprostredkovateľov úlohu ovládača (controller). Ten je zvolený aktívnymi sprostredkovateľmi klastru. Takýto sprostredkovateľ má na starosti napríklad monitorovanie zlyhania iných sprostredkovateľov alebo priraďovanie partícií.




Obr. 5: Znázornenie Kafka klastru s troma sprostredkovateľmi.

Kafka je určená k tomu, aby pracovala ako klaster nakoľko z toho plynie viacero výhod či už z hľadiska škálovateľnosti alebo dostupnosti. Čo sa partícií týka, tak každá z nich má svojho lídra (leader). Partícia však môže (ale nemusí) byť priradená viacerým sprostredkovateľom, avšak je nevyhnutné, aby boli spojení s lídrom (pričom partícia má stále iba jedného z nich ako lídra). V takomto prípade sa stane to, že partícia bude replikovaná a ak nastane prípad, že by bol líder danej partície nedostupný môže ho zastúpiť iný sprostredkovateľ.[5]

Kafka však k svojej činnosti potrebuje Apache Zookeeper. Apache Zookeeper je open-source softvér, ktorý poskytuje riešenia pre rôzne koordinačné problémy vo veľkých distribuovaných systémoch.[8] Apache Kafka používa Zookeeper na ukladanie metadát ohľadom

[3]Takýto počet však môže byť užitočný pokiaľ sa odpojí niektorí z odberateľov nakoľko príde k prerozdeleniu partícií.

Kafka klastru. Zookeeper si drží informácie nielen o jednotlivých uzloch Kafka klastru, ale taktiež aj o všetkých témach, partíciách a tak ďalej.

## 1.3 Kafka API

Apache Kafka poskytuje štyri aplikačné rozhrania, ktoré je možné použiť pri práci s touto platformou. Konkrétne sú to:

- Producer API
- Consumer API
- Streams API
- Connector API
- Admin API

**Producer API** alebo aj vydavateľské aplikačné rozhranie slúži k publikovaniu správ resp. záznamov do jednotlivých tém v Apache Kafka.

**Consumer API**, teda odberateľské rozhranie, sprístupňuje možnosť aplikácií odoberať z jednotlivých tém.

**Streams API** sa využíva ako aplikačné rozhranie na spracovávanie tokov dát. Aplikácia vystupuje ako spracovateľ tokov dát, kde môže napríklad konzumovať nejaký vstupný dátový tok z témy a vyprodukovať výstupný dátový tok pre inú tému. Taktiež je tento proces možný aj pre viacero tém nielen pre jednu. Toto rozhranie umožňuje taktiež rôzne manipulácie s dátami, ktoré sa nachádzajú v tokoch.

Predposledným z týchto rozhraní je **Connector API**, ktoré slúži ako nástroj k presunu veľkých kolekcií dát do Kafky a taktiež aj z nej. Je možné týmto rozhraním napríklad zachytiť každú zo zmien databázovej tabuľky prípadne zberať metriky z rôznych aplikačných serverov a tak ďalej. [6]

Posledné z kolekcie rozhraní predstavuje **Admin API**. Toto rozhranie slúži k spravovaniu a inšpekcií jednotlivých tém, sprostredkovateľov a iných Kafka objektov.

Treba však brať na ohľad, že spomenuté aplikačné rozhrania predstavujú základné rozhrania. Apache Kafka má však vybudovaný celý ekosystém. Kafka má k dispozícií rozšírenia ako napríklad Kafka Connect, ktorý poskytuje možnosť integrácie Apache Kafka s inými systémami ako napríklad GCP Pub/Sub, Kinesis a mnohými ďalšími.

## 1.4 Výhody Apache Kafka

Na záver by sme chceli poukázať na niektoré silné stránky Apache Kafka, ktoré je možné zvážiť pri výbere sytému na spracovávanie dátových zdrojov.

**Odolnosť voči chybám**

Jednou z výhod Apache Kafka je jej odolnosť voči poruchám. Kafka vie riešiť zlyhanie uzla v klastri práve tým, že dokáže zabezpečiť replikáciu jednotlivých partícií témy. Túto replikáciu je možné ovplyvniť nastavením replikačného faktora.

**Škálovateľnosť**

Flexibilná škálovateľnosť Kafky uľahčuje manipuláciu s akýmkoľvek množstvom údajov. Užívatelia môžu začať s jedným sprostredkovateľom ako prototypom svojej aplikácie. Následne rozšíriť aplikáciu na malý vývojový klaster s troma sprostredkovateľmi a potom presunúť aplikáciu do produkčného prostredia s väčším klastrom s desiatkami alebo dokonca stovkami sprostredkovateľov.

**Disková retencia**

Správy v Kafke sa ukladajú na disk a sú tam uchovávané podľa nakonfigurovaných pravidiel, ktoré sa vzťahujú vždy na každú tému osobitne. Možno použiť buď to uchovávanie s časovou exspiráciou, kedy používateľ nastaví po aký dlhý čas majú byť správy v Kafke uchované kým sa vymažú (napr. jeden týždeň). Alebo je možné zvoliť exspiráciu na základe množstva uložených údajov, kedy používateľ zvolí po akom množstve dát sa majú správy vymazať (napr. 5 gigabajtov). Takýto spôsob retencie taktiež zabezpečí, že pokiaľ je odberateľ, ktorý nestíha konzumovať správy, tak si ich v Kafke nájde i napriek svojmu oneskoreniu.[5]

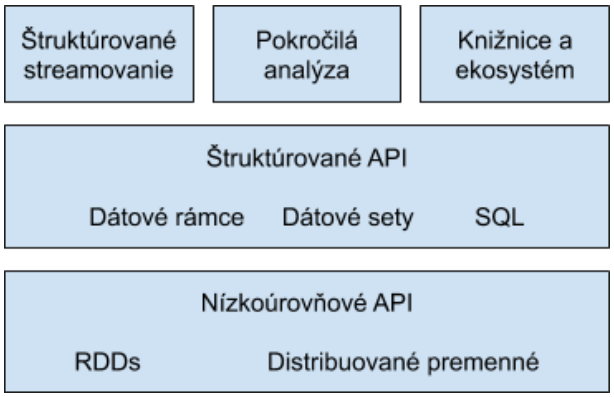
# 2 Apache Spark

Apache Spark je unifikovaný výpočtový nástroj spolu so sadou knižníc pre paralelné spracovanie údajov v počítačových klastroch.[9] Spark je nástroj na prácu s veľkým obnosom dát, ktorý je využívaný ako programátormi tak i dátovými inžiniermi (data engineer). Svoju obľubu si získal aj vďaka širokému portfóliu programovacích jazykov, ktoré podporuje ako napríklad Java, Scala či Python. Avšak taktiež podporuje i programovací jazyk R a má aj knižnice, ktoré umožňujú využívať SQL (štruktúrovaný dopytovací jazyk). Spark taktiež sprístupňuje knižnice, ktorými je možné pracovať s dátovými tokmi prípadne so strojovým učením.

Vývoj Apache Spark začal na University of California, Berkeley skupinou AMPLab v roku 2009. Je dôležité poznamenať, že v tomto čase dominoval Apache Hadoop, ktorý využíva MapReduce slúžiaci na spracovanie veľkého množstva dát paralelne naprieč klastrom, ktorý môže pozostávať z veľkého množstva uzlov. MapReduce rozdelí množinu vstupných údajov na nezávislé časti, ktoré sú následne spracovávané úplne paralelným spôsobom pomocou mapovania. Výstupy týchto mapovacích operácií resp. úloh následne potom prechádzajú redukčnými operáciami. Vstupy aj výstupy úloh sa zvyčajne odkladajú na disk (v prípade Apache Hadoop je to HDFS).

Ľudia pracujúci na vývoji Apache Spark pracovali a komunikovali s používateľmi MapReduce a rozumeli benefitom rovnako ako aj nevýhodám tejto technológie. Preto ich cieľom bolo využiť potenciál klastrových výpočtov. Avšak taktiež chceli odstrániť nevýhodu, ktorú so sebou prinášal MapReduce, ktorý napríklad v prípade niektorých algoritmov strojového učenia mohol potrebovať niekoľko násobný prechod dátami. Treba však myslieť na to, že v prípade viacerých prechodov prichádza k poklesu výkonu nakoľko vstup a taktiež aj výstup takéhoto prechodu musí byť načítaný prípadne zapísaný na disk.

Toto sú problémy a požiadavky, ktoré sa snažili tvorcovia Spark-u vyriešiť. Preto napríklad v Spark-u využili nový mechanizmus, ktorý umožnil vykonávať efektívne zdieľanie údajov v pamäti (RAM) pri viacerých iteráciách. Spočiatku bolo možné vytvárať iba aplikácie, ktoré pracovali nad dávkovanými dátami avšak postupom času sa rozhodli autori pridať aj podporu pre interaktívne dotazy. Neskôr začali do Apache Spark pribúdať rôzne knižnice ako napríklad MLib (ktorý slúži na vývoj Spark aplikácií s prvkami strojového učenia) alebo Spark Streaming (knižnica umožňujúca prácu s tokmi dát) a mnohé ďalšie knižnice, ktoré pribudli v neskoršej fáze vývoja a momentálne tvoria neoddeliteľnú súčasť Apache Spark. To ako momentálne vyzerá rozdelenie jednotlivých komponentov do aplikačných rozhraní je zobrazené na obrázku 6.

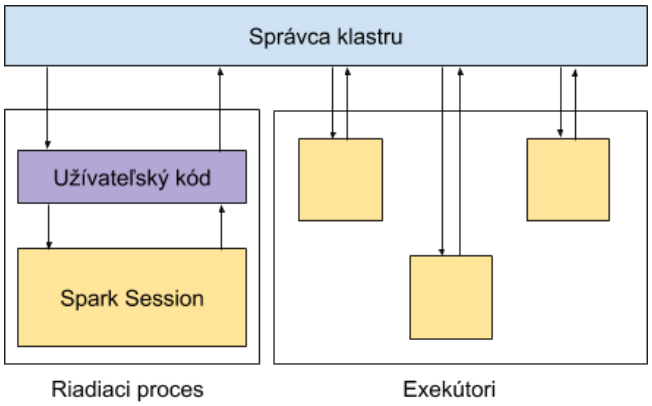



Obr. 6: Rozdelenie jednotlivých komponentov Apache Spark[9]

## 2.1 Architektúra Apache Spark

Jednou z výhod Apache Spark je jeho možnosť činnosti tak ako v lokálnom móde (napríklad na notebooku) tak aj v klastri. Klaster je však preferované prostredie, v ktorom je Apache Spark využívaný. K činnosti v klastri je možné využiť jeden z troch klastrových správcov. Okrem natívneho správcu je možné využiť Hadoop YARN prípadne Apache Mesos. Apache Spark je možné od verzie 2.4.5 používať aj na klastri, ktorý je spravovaný pomocou Kubernetes. Je však potrebné poznamenať, že táto možnosť je zatiaľ označená ako experimentálna.

To znamená, že v rámci klastru môže byť spustených hneď niekoľko Spark aplikácií súčasne, nie iba jedna. V praxi je potrebné iba zaslať Spark aplikáciu príslušnému klastrovému správcovi, ktorý sa postará o sprístupnenie prostriedkov pre danú aplikáciu k tomu, aby mohla splniť svoju prácu.




Obr. 7: Architektúra štandardnej Apache Spark aplikácie

Architektonicky Spark aplikácia pozostáva z troch častí: riadiaceho procesu, správcu klastru a exekútorov. Riadiaci proces je komponent, ktorý má na starosti beh hlavnej

metódy aplikácie. Je zodpovedný primárne za nasledovné tri úlohy.

Prvou z nich je spravovanie údajov o aplikácií. Následne má na starosti spracovanie inštrukcií užívateľského kódu prípadne spracovanie vstupov. V neposlednom rade je jeho povinnosťou spravovanie exekútorov. To zahŕňa činnosti od analýzy ich stavu, prerozdeľovaniu práce až po následné zaslanie jednotlivých úloh exekútorom.

Riadiaci proces je ústrednou časťou aplikácie, ktorá riadi celý jej chod od začiatku až po dokončenie všetkých požadovaných inštrukcií.

Ďalšou z častí architektúry sú už spomínaní exekútori. Je dôležité uviesť, že exekútorov môže byť ľubovoľné množstvo na danom uzle a toto množstvo je uvedené v konfigurácií. Tak ako riadiaci proces tak aj každý exekútor je iba proces. V prípade lokálneho režimu môžu bežať ako jednotlivé vlákna. V prípade klastru títo exekútori ani nemusia byť na tom istom zariadení. To práve z toho dôvodu, že ich koordináciu má na starosti riadiaci proces.

Exekútor ako taký má na starosti dve veci. Prvá z nich je vykonávanie úloh, ktoré sú mu zaslané na vykonanie riadiacim procesom. Jeho druhou úlohou je podávať hlásenia o stave výpočtu, ktorý vykonáva. Správca klastru má za úlohu prerozdelenie prostriedkov klastru, ktoré má v daný moment k dispozícií.

## 2.2 Porovnanie programovacích jazykov

Jazykové aplikačné rozhrania Apache Spark umožňujú používanie rôznych programovacích jazykov. Program je v Apache Spark možné písať v nasledujúcich programovacích jazykoch: Java, Scala, Python a R. Avšak mimo tieto jazyky Spark poskytuje podporu aj pre SQL, ktorá bude opísaná nižšie.

### 2.2.1 JVM a Non-JVM jazyky

Vymenované jazyky, ktoré Apache Spark podporuje je možné rozdeliť do dvoch kategórií nasledovne:

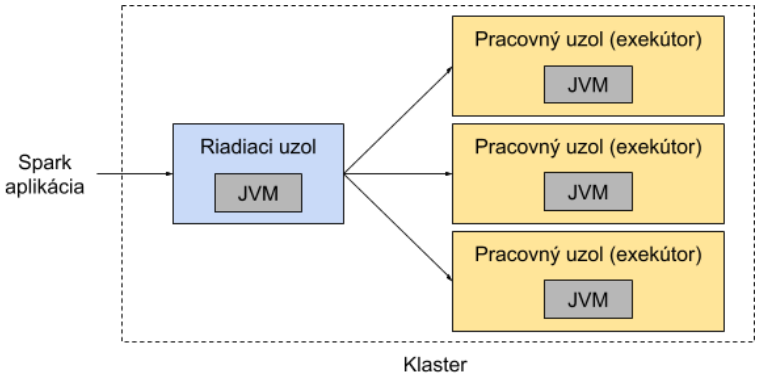
- JVM jazyky - Java a Scala
- Non-JVM jazyky - Python a R

Táto kategorizácia bude nápomocná v neskoršom porovnávaní. Taktiež je potrebné dodať, že Apache Spark je napísaný v jazyku Scala. Zdrojový kód napísaný v jazyku Scala je určený na kompiláciu do Java bajt kódu (Java bytecode), takže výsledný spustiteľný kód je následne možné spustiť v JVM.

Navyše Scala poskytuje jazykovú interoperabilitu s jazykom Java. Takže knižnice napísané v jazyku Scala alebo Java sa môžu na seba odkazovať priamo v kóde jedného z

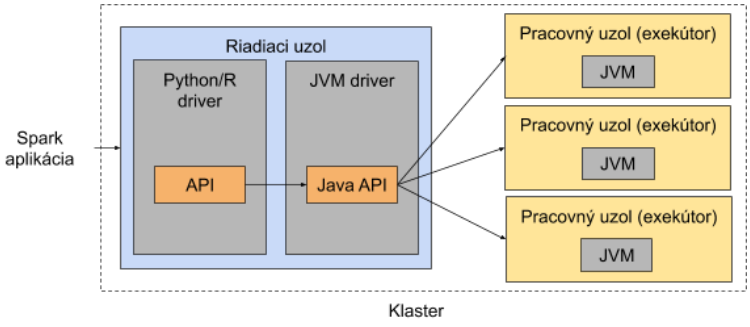
nich. Z toho teda vyplýva, že aplikácie Apache Spark, nakoľko je napísaný v jazyku Scala, je možné spustiť v JVM bez žiadnej konverzie, ktorá by mohla ovplyvňovať výkon. To znamená, že jazyky zo skupiny JVM z pôvodného delenia by na tom mali byť výkonovo lepšie ako jazyky z non-JVM nakoľko nepotrebujú dodatočnú konverziu v žiadnom bode vykonávania programu.

Treba však brať na ohľad spôsob vykonávania kódu v prípade oboch skupín. V prípade kategórie JVM jazykov je Spark aplikácia nahraná do riadiaceho uzlu, ktorý iba distribuuje jednotlivé úlohy pracovným uzlom (worker node), kde v prípadoch oboch typov uzlov sa jedná o spustenie kódu priamo v JVM viď obr. 8.



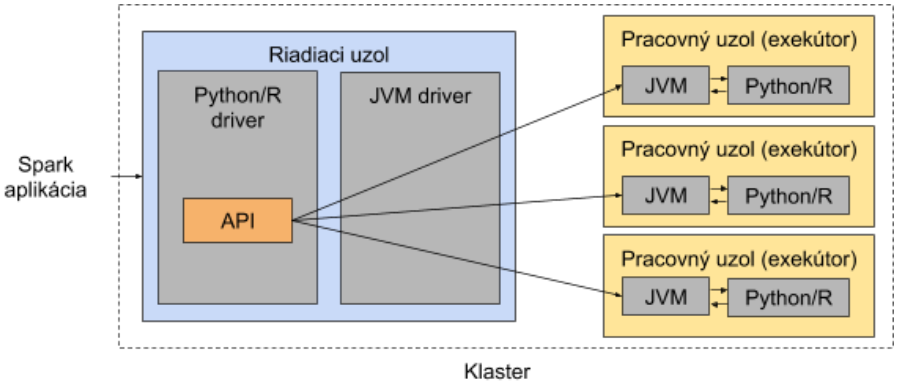

Obr. 8: Vykonávanie aplikácie pri JVM kompatibilnom jazyku

V takomto prípade je možné cenu medziprocesnej komunikácie z hľadiska výkonu v porovnaní s JVM jazykmi zanedbať (obr. 9). Avšak to iba za predpokladu, že sa spracované dáta z pracovných uzlov nebudú vracať späť do riadiaceho uzla. Ak by bolo potrebné ich vrátenie do riadiaceho uzla tak v takom prípade by sa značne navýšili režijné náklady a taktiež by bola nutná serializácia, čo by značne zvýšilo čas spracovania, a preto by sa zhoršila celková výkonnosť v takomto scenári.




Obr. 9: Vykonávanie aplikácie pri non-JVM kompatibilnom jazyku v prípade existencie ekvivalentnej funkcie

Potom je tu druhý scenár, ktorý nastáva v momente, kedy neexistuje ekvivalent v Java/Scala riadiacom API (obr. 10). Takýto prípad môže nastať napríklad v prípade užívateľom zadefinovaných funkciách alebo v prípade použití Python knižníc tretích strán.




Obr. 10: Vykonávanie aplikácie pri non-JVM kompatibilnom jazyku v prípade, že neexistuje ekvivalentná funkcia

V takomto prípade Spark spustí Python proces na každom jednom z pracovných uzlov, zoserializuje všetky dáta do formátu, ktorý je zrozumiteľný pre Python (nakoľko sa pôvodne jednalo o JVM), vykoná samotnú funkciu v Python procese a následne vráti výsledok do JVM a Apache Spark.[9]

Z hľadiska výkonu je potrebné brať na ohľad náročnosť resp. cenu vytvorenia Python procesu. To je samozrejme dôležitý faktor avšak kľúčovým faktorom v celom tomto procese je samotná serializácia dát, ktorú je pre Python potrebné uskutočniť. Jeden z dôvodov je ten, že takáto serializácia je náročná z hľadiska výpočtu.

Taktiež je však potrebné zdôrazniť to, že po tom, ako sú serializované dáta úspešne zaslané, Spark nemôže spravovať pamäť daného Python procesu, ktorý vykonáva operácie, čo v konečnom dôsledku môže vyústiť do súperenia o pamäť medzi JVM a Python-on a následne skončiť zlyhaním procesu z dôvodu takéhoto súperenia o prostriedky.

### 2.2.2 Čitateľnosť

Ďalším dôležitým faktorom pri porovnávaní jazykov je čitateľnosť kódu. Dlhšiu dobu prevládal názor, že Scala a Python mali lepšiu podporu v API v porovnaní s programovacím jazykom Java. Dôvod bol predovšetkým ten, že v jazyku Java chýbali funkčné vyjadrenia (function expression), ktoré z tohto dôvodu značne komplikovali kód nie moc prehľadnými anonymnými vnútornými triedami, ktoré bolo potrebné vytvárať pri práci s API.

Tento nedostatok bol však odstránený v Java verzii 8 a následnou aktualizáciou zo

strany Spark verzie 1.0. V tejto verzii bola pridaná podpora pre lambda výrazy, ktoré priniesla Java 8.0 avšak stále zostala podpora predošlého prístupu, aby mohli byť i naďalej použité staršie verzie jazyku Java.

```
JavaRDD lines = sc.textFile("hdfs://log.txt").filter(
  new Function() {
    public Boolean call(String s) {
      return s.contains("error");
    }
});
long numErrors = lines.count();
```

Kód 1: Ukážka Spark kódu v jazyku Java 7

```
JavaRDD lines = sc.textFile("hdfs://log.txt").filter(s -> s.contains("error"));
long numErrors = lines.count();
```

Kód 2: Ukážka Spark kódu v jazyku Java 8

Bez potreby znalostí fungovania Apache Spark je možné vidieť v čitateľnosti, kedy sa oproti verzii 7 (viď. kód 1) nevyžaduje vo verzii 8 (viď. kód 2) tvorba triedy *Function* definovaním metódy *call* a použitím lambda funkcie je možné už iba zavolať samotnú metódu *contains*. Takáto zmena možno nemusí pôsobiť zvlášť závažne pri malej ukážke kódu, kedy sa zredukuje počet riadkov zo sedem na dva avšak pri väčších aplikáciách, ktoré majú na starosti komplexnejšie transformácie dát a logiku to dokáže spraviť omnoho väčší rozdiel.

Čo sa týka nasledujúceho porovnávania čitateľnosti kódu budú použité ukážky v jazykoch Scala, Java (vo verzii 8) a Python. Pri ukážkach nie je dôležité, čo má daný kód docieliť a taktiež v nich nie sú uvádzané parametre funkcií[4], ktoré slúžia iba konfiguračným účelom a nemajú vplyv na čitateľnosť.

Napriek tomu, že každá z ukážok vykonáva v konečnom dôsledku tú istú funkcionalitu, v každom jazyku sú isté odlišnosti toho, ako je tento cieľ dosiahnutý. Na úvod porovnania je dôležité upozorniť na jeden z mnohých rozdielov jazykov Java a Scala v porovnaní s jazykom Python. Java a Scala sú totižto radené medzi staticky písané jazyky (statically typed) kým Python je zaradený do skupiny dynamicky písaných jazykov (dynamically typed).

[4]Tieto parametre sú nahradené v kóde symbolom „...“

**Staticky písané jazyky** sa vyznačujú tým, že typ premennej, ktorá je v programe použitá je známy už v momente kompilácie v kontraste s **dynamicky písanými jazykmi**, ktorých typ premenných je známy až počas behu programu.

Avšak aj samotné jazyky, ktoré sú staticky písané sa ešte môžu líšiť, čo je aj prípad jazykov Java a Scala. Scala totižto využíva odvodenie typu (type inference), čo je schopnosť jazyka vydedukovať typ premennej bez potreby ho explicitne uvádzať. Z toho vyplýva, že v jazyku Java je potrebné uviesť o aký typ premennej sa jedná, zatiaľ čo Scala s využitím odvodenia typu ho vie vydedukovať a tým pádom nie je potrebné ho uvádzať. Preto už na prvý pohľad sú Scala a Python jednoduchšie čitateľné nakoľko neodvádzajú pozornosť čitateľa zdĺhavými typmi premenných ako je to v prípade jazyku Java.

```
int numStreams = 5;
List<JavaPairDStream<String, String>> kafkaStreams = new ArrayList<>(numStreams);
for (int i = 0; i < numStreams; i++) {
  kafkaStreams.add(KafkaUtils.createStream(...));
}
JavaPairDStream<String, String> unifiedStream =
    streamingContext.union(kafkaStreams.get(0), kafkaStreams.subList(1,
    kafkaStreams.size()));
unifiedStream.print();
```

Kód 3: Ukážka kódu v jazyku Java

```
val numStreams = 5
val kafkaStreams = (1 to numStreams).map { i => KafkaUtils.createStream(...) }
val unifiedStream = streamingContext.union(kafkaStreams)
unifiedStream.print()
```

Kód 4: Ukážka kódu v jazyku Scala

```
numStreams = 5
kafkaStreams = [KafkaUtils.createStream(...) for _ in range (numStreams)]
unifiedStream = streamingContext.union(*kafkaStreams)
unifiedStream.pprint()
```

Kód 5: Ukážka kódu v jazyku Python

Tento fakt je možné si všimnúť napríklad pri vytváraní premennej *kafkaStreams*, ktorá je v prípade jazykov Scala a Python inicializovaná príslušnými hodnotami volania *createStream* v cykle použitím jedného riadku kódu. Na druhej strane v Java-e je táto úloha trochu menej prehľadná nakoľko bola najprv použitá deklarácia premennej sprevádzaná dlhým názvom typu *List<JavaPairDStream<String, String»* a následným vyplnením hodnôt pomocou tradičného for-cyklu.

Taktiež je možné si všimnúť odlišnosť vo volaní metódy *union*, ktorej vstupné parametre sú veľmi podobné v prípade jazykov Python a Scala a rozdielne v prípade jazyku Java. Práve aj takéto mierne odlišnosti v API prípadne funkciách konkrétneho jazyka môžu mať vplyv na celkovú čitateľnosť a prehľadnosť kódu.

### 2.2.3 Iné faktory

Čo sa týka porovnania jednotlivých programovacích jazykov je potrebné taktiež poukázať i na ďalšie faktory ktoré môžu ovplyvniť vývoj aplikácií v Apache Spark. Rozhodovanie v zmysle, či zvoliť jazyk Java alebo Scala môže byť o preferencií konkrétneho programátora a o tom, akú syntax preferuje, prípadne s ktorým z JVM jazykov má viac skúseností.

Apache Spark však môžu používať aj ľudia, ktorí nie sú programátori. Medzi týchto ľudí môžu patriť napríklad dátoví analytici, IT biznis analytici alebo ľudia, ktorí pracujú predovšetkým v oblasti umelej inteligencie a neprichádzajú do styku resp. nemajú skúsenosti s programovaním v jazyku Java prípadne Scala. Takéto skupiny ľudí však taktiež môžu využívať Spark.

SQL, alebo aj štruktúrovaný vyhľadávací jazyk, slúži na manipuláciu s dátami uloženými v databázach. Apache Spark podporuje podmnožinu *ANSI SQL 2003*. To môže pomôcť vyššie spomínaným ľuďom ako sú analytici prípadne ľudia, ktorí majú skúsenosti s platformou Oracle alebo ľudia, ktorí nemajú skúsenosti s programovaním. Pokiaľ však majú skúsenosti s SQL tak potom môžu využívať výhody, ktoré Spark ponúka pri práci s veľkým obnosom dát. V Spark SQL je možné využiť tak ako analytické funkcie, tak i agregačné funkcie a navyše bola do Apache Spark od verzie 1.4 pridaná podpora aj pre *Window Functions*.[10]

Výhoda použitia Spark SQL je taktiež v tom, že jednotlivé dopyty, ktoré sú v ňom napísané sa dajú veľmi jednoducho prepoužiť aj pri zmene programovacích jazykov. Je tým pádom jedno, či je kód Apache Spark písaný v jazyku Java, Python alebo inom, nakoľko SQL dopyt bude vo všetkých prípadoch rovnaký.

Na ukážkach kódu 6 a kódu 7 je možné vidieť ako sa dá docieliť tá istá funkcionalita použitím SQL a použitím metód z *Dataset API*. Obe tieto ukážky v konečnom dôsledku

vyprodukujú ten istý výsledok. V prípade Spark SQL sme použili jeden SQL dopyt a v druhom prípade sme použili volanie piatich metód. V tomto prípade môže práve kód 6 pôsobiť čitateľnejšie a taktiež môže byť aj ľahšie udržiavateľný v porovaní s kódom 7.

```
val destMax = spark.sql("""
  SELECT dest_country_name, SUM(count) as total_destinations
  FROM flight_data
  GROUP BY dest_country_name
  ORDER BY SUM(count) DESC
  LIMIT 10
""")
destMax.collect()
```

Kód 6: Ukážka kódu s použitím Spark SQL

```
import org.apache.spark.sql.functions.desc
flightData
  .groupBy("dest_country_name")
  .sum("count")
  .withColumnRenamed("sum(count)", "total_destinations")
  .sort(desc("total_destinations"))
  .limit(10)
  .collect()
```

Kód 7: Ukážka kódu s použitím metód z Dataset API

Druhou z týchto skupín môžu byť ľudia pracujúci s umelou inteligenciou. Takáto skupina ľudí bude s veľkou pravdepodobnosťou uprednostňovať používanie jazyku Python. Je to práve z toho dôvodu, že Python je jeden z najpoužívanejší jazykov v oblasti umelej inteligencie. Svedčiť o tom môže aj podpora veľkej komunity vyvíjajúcej rôzne knižnice a nástroje pre tento programovací jazyk. Jedná sa napríklad o knižnice slúžiace priamo na vývoj strojového učenia ako napríklad Keras, TensorFlow, Scikit-learn a mnohé ďalšie. Avšak i knižnice na dátovú analýzu a vedecké výpočty ako NumPy alebo Pandas a ďalšie knižnice na vizualizáciu dát ako Matplotlib či Plotly. Práve podpora týchto (prevažne open-source) nástrojov pomáha jazyku Python v obľúbenosti medzi ľuďmi, ktorí pracujú v oblasti umelej inteligencie. Práve preto tvorcovia Apache Spark prinášajú podporu pre Python v podobe PySpark.

## 2.3 Základné koncepty

Pre pochopenie fungovania Apache Spark je potrebné poznať tak ako abstrakcie, ktoré Spark využíva, tak aj spôsob akým nad týmito abstrakciami pracuje. V Apache Spark sú tieto abstrakcie rozdelené do dvoch API pričom je na používateľovi, ktoré z nich bude využívať. Do istej miery je dokonca možné ich aj kombinovať. Následne je potrebné vedieť s dátami narábať, filtrovať ich prípadne ich istým spôsobom pozmeniť. Na tento typ operácií Spark využíva takzvané transformácie. V neposlednom rade býva často potrebné dáta niekam odoslať prípadne ich perzistovať na nejakom úložisku a to už či sa jedná o databázu alebo súborový systém. Jednotlivé abstrakcie a vnútorné fungovanie Spark-u je do väčších detailov popísané v nasledujúcej kapitole.

### 2.3.1 Aplikačné rozhrania

Apache Spark poskytuje developerom tri základné rozhrania, ktoré môžu byť využité pri práci s dátami. Jedná sa konkrétne o RDD, čo je skratka pre Resilient Distributed Dataset alebo voľne preložené ako odolný distribuovaný dátový set resp. množina údajov. RDD ako možno vidieť na obrázku 6 je súčasťou nízkoúrovňového API a bolo súčasťou Apache Spark od samotného vzniku. Následne pribudlo rozhranie označované ako DataFrame alebo dátový rámec[11]. Toto rozhranie bolo do Spark-u pridané vo verzií 1.3 s cieľom pridať rozhranie, ktoré by oslovilo škálu potenciálnych užívateľov aj mimo big data inžinierov pomocou rozhrania, ktoré je viacerým vývojárom známe. Ako momentálne najaktuálnejšie rozhranie vystupuje API s označením DataSet alebo dátový set. Toto rozhranie bolo prvý krát predstavené v Spark verzií 1.6 a vyvíjané od spomínanej verzie až po súčasnosť. V nasledujúcich kapitolách popíšeme tieto jednotlivé rozhrania podrobnejšie a predstavíme na nich základnú filozofiu Apache Spark pri práci s dátami.

**RDD** Každá aplikácia v Apache Spark pozostáva z riadiaceho procesu, ktorý má na starosti spustiť a riadiť vykonávanie užívateľom definovanej funkcie známej taktiež aj ako *main*. Tento riadiaci proces kontroluje chod Spark aplikácie a je pomenovaný ako SparkSession. Je to v podstate spôsob akým možno pomocou inštancie tejto SparkSession vykonávať príkazy a manipulácie s dátami definované používateľom.

RDD, ako je popisované v práci zakladateľov Apache Spark[12], predstavuje distribuovanú abstrakciu pamäte, ktorá dokáže programátorom umožniť vykonávanie rôznych výpočtov priamo v pamäti či už počítača alebo výkonného klastru a to pri zachovaní odolnosti voči poruchám. Okrem tejto vlastnosti predstavuje RDD nemennú paralelnú dátovú štruktúru[13]. RDD umožňuje používateľom udržiavať priebežné výsledky priamo v pamäti a taktiež riadiť ich rozdelenie a manipulovať nimi.

Je dôležité zdôrazniť nemennosť tejto štruktúry. Tento pojem je v súvislosti s programovaním často skloňovaný. V prípade dátových štruktúr sa jedná o nemennosť v tom zmysle, že od momentu ich vzniku nemôžu byť zmenené[14]. Jednou z výhod nemenných dát je to, že môžu byť bezpečne zdieľané medzi procesmi, čo je presne spôsob, ktorý zvolili tvorcovia Apache Spark pre vyriešenie problému odolnosti voči chybám pri zachovaní efektívnosti.

Okrem toho nie sú len nemenné, ale navyše sú aj deterministickými funkciami vzhľadom na vstup. To spolu s nemennosťou znamená, že dáta vedia byť znovu vytvorené v ľubovoľnom čase, čo umožňuje zabezpečiť odolnosť voči chybám logovaním jednotlivých zmien, ktoré boli uskutočnené nad dátami miesto logovania samotných dát ktoré boli menené. Hlavnou výhodou je to, že takáto dátová štruktúra vie byť ľahko obnoviteľná a tým pádom môže byť zrýchlené spracovanie, nakoľko nie je potrebná replikácia medzi uzlami klastru.

S myšlienkou ako by sa dal zrýchliť výpočet nad veľkým obnosom údajov uložených na disku použitím paralelizmu prišli výskumníci z Google. Je odporúčané si preštudovať tento prístup nakoľko v tejto práci sa nebudeme detailne venovať spôsobu fungovania MapReduce. Je však potrebné vedieť, že pomocou MapReduce je možné veľmi efektívne pracovať napríklad nad veľkým množstvom súborov, ktoré sú uložené na disku. Nakoľko, čo interne robí MapReduce je prečítanie a namapovanie údajov, uloženie údajov, prečítanie a redukovanie údajov a následné uloženie. Avšak tento spôsob má svoje nevýhody napríklad v prípade iteratívnych algoritmov. Tie môžu predstavovať napríklad algoritmy strojového učenia, ktoré sú úzko späté s iteráciami. To by vyžadovalo neustále opakovanie vyššie spomenutého opakovania prístupu na disk. Operácie s diskovým médiom sa však radia medzi pomalé operácie, ktoré v takomto prípade značne ovplyvňujú celkový výkon[15][16].

Tvorcovia Apache Spark však zaviedli výpočet v pamäti. Dáta sú tak držané v RAM teda pamäti s náhodným prístupom. RDD sa tak pri výpočte ukladajú do pamäte, čím sa eliminujú práve drahé operácie (z hľadiska výpočtového času) ako zápis na disk[17].

**Dátové sety a dátové rámce** Dátové sety a rámce, na rozdiel od RDD, patria do štruktúrovaného aplikačného rozhrania. Pomocou nich môžu byť pomerne jednoducho reprezentované dáta, ktoré si možno predstaviť ako tabuľku či už z oblasti databáz alebo napríklad aj z oblasti tabuľkových procesorov, ktoré s tabuľkami pracujú.

Dáta sú reprezentované pomocou stĺpcov a riadkov. V tomto bode sa môže vyskytnúť otázka, prečo vôbec použiť Apache Spark pokiaľ existuje toľko podobností v rovine abstrakcie v porovnaní s tabuľkovými procesormi. V prípade takýchto tradičných procesorov

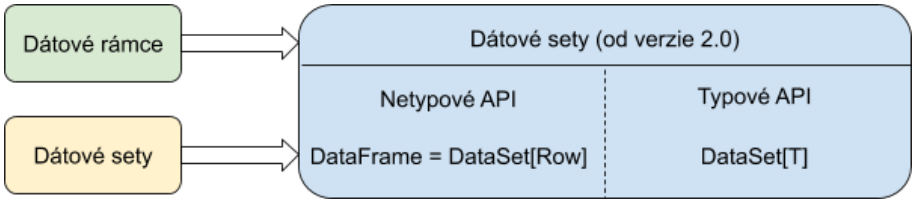
býva zvykom, že takáto tabuľka je umiestnená lokálne na jednom zariadení ako napríklad počítač, v prípade takéhoto dátového rámcu v Apache Spark môže byť tento rámec resp. set rozdistribuovaný po niekoľkých počítačoch.

Z toho plynie hneď niekoľko výhod nakoľko v prípade tabuľkového procesoru môže nastať situácia, kedy bude za potreby načítať tabuľku, ktorá sa nebude schopná zmestiť do pamäte daného počítača, na ktorom beží takýto procesor. Taktiež môže nastať aj prípad, kedy by sa tabuľka síce mohla zmestiť do pamäte počítača avšak trvalo by príliš dlho previesť na nej požadované operácie.

Pri Apache Spark je však možné použiť dátový rámec prípadne set. V takom prípade je veľkosť pamäte omnoho menšou limitáciou nakoľko Spark môže byť spustený na klastri a navyše dokáže dáta rozdeliť medzi uzli klastru. Taktiež operácie Apache Spark, ktoré bude používateľ chcieť vykonať môžu byť vykonávané paralelne, čo z neho robí vhodným riešením pre práve takýto väčší obnos dát.

Koncept dátového rámcu však nepoužíva iba Apache Spark. Podobný koncept vedia využiť i iné programovacie jazyky, medzi ktoré patrí napríklad aj Python. Tieto rámce však v prípade Python[5] zväčša majú podobné obmedzenie ako už spomínané tabuľky v prípade tabuľkových procesorov a je na nich operované výhradne v rámci jedného počítača. Spark však využitím svojej API dokáže dátové rámce používané v jazyku Python (prípadne aj v jazyku R) previesť na svoje dátové rámce a získať tak prístup ku všetkým výhodám, ktoré prinášajú.

Dôvod, prečo opisujeme rámce spolu so setmi v jednej sekcií je práve ten, že dátový set spolu s dátovým rámcom, ako uvádzajú tvorcovia Apache Spark, už nie sú naďalej dve odlišné API nakoľko boli od verzie 2.0.[6] zjednotené[18].




Obr. 11: Unifikácia dvoch aplikačných programových rozhraní v Apache Spark

Kľúčovým rozdielom medzi rámcom a setom je práve spôsob narábania s typmi. Ako je možné z obrázku 11 vidieť, dátové sety v netypovom API obsahujú položky typu

[5]Existujú Python knižnice ako napríklad Dask, ktoré dokážu distribuovať rámec naprieč klastrom avšak natívne v jazyku Python nie je táto možnosť.

[6]Ako je možné si z obrázku všimnúť DataFrame je stále používaným aliasom pre dátový set obsahujúc typ *Row* alebo riadok.

*Row*. *Row* je všeobecný riadkový objekt s usporiadanou zbierkou položiek, ku ktorým je možné získať prístup pomocou indexu, názvu alebo pomocou porovnávania vzorov v jazyku Scala[19]. Na druhej strane typové API ponúka možnosť v podobe DataSet[T], kde T predstavuje užívateľom zvolený objekt. Výhoda použitia typu je práve v tom, že táto API vyjadruje všetko v podobe lambda funkcií a JVM typových objektov, z čoho plynie benefit pre vývojárov práve v tom, že všetky syntaktické chyby v podobe nesúladu typov budú detegované ešte v čase kompilácie.

Spomínaný paralelizmus týchto rámcov dosahuje Spark za pomoci partícií, do ktorých tieto rámce rozdeľuje. Partíciu je možné si predstaviť ako konkrétnu časť tabuľky, ktorá sídli v počítači. Častou sa v tomto prípade myslí nejaké množstvo riadkov. Partícia vo svojej podstate reprezentuje rozdelenie dát na jednotlivých zariadeniach v klastri. Nakoľko rámce nepatria medzi nízkoúrovňovú API, používateľ nemá na starosti manipuláciu a spravovanie jednotlivých partícií[7]. Programátorsky dátové rámce predstavujú istú kolekciu objektov, ktoré sú typu stĺpec. Každý z týchto stĺpcov má svoje pomenovanie, vďaka ktorému k nemu dokáže používateľ pristupovať a konceptuálne je celý tento systém porovnateľný s databázovými tabuľkami.

### 2.3.2 Operácie

V predchádzajúcich kapitolách sme si predstavili nemenný charakter štruktúry RDD. Apache Spark plne využíva túto vlastnosť pri práci s týmito štruktúrami. Úkony, ktoré sa používajú na narábanie s dátami sa označujú ako operácie. Operácie v Apache Spark sa delia do dvoch základných kategórií:

- transformácie,
- akcie.

**Transformácie** Transformácie predstavujú typ operácie v Apache Spark, ktorý popisuje to, ako budú dáta transformované resp. ako budú dáta upravené, pozmenené a tak ďalej. Transformáciu si môžeme predstaviť ako funkciu, ktorej vstup predstavuje RDD, vykoná na nej užívateľom zvolené úpravy a jej výstupom bude nová RDD, ktorá bude obsahovať dané zmeny. Veľmi dôležité je v tomto prípade výraz „nová RDD“ nakoľko ako sme popísali vyššie RDD má nemenný charakter, ktorý sa samozrejme aj v tomto prípade zachováva a tým pádom pri transformácii vždy vznikne nová RDD[8].

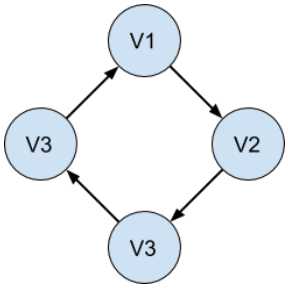
[7]Je možné narábať s partíciami aj pomocou dátových rámcov, avšak nie je to nevyhnutné až do momentu, kedy používateľ chce ladiť výkon.

[8]Aj táto novovzniknutá RDD je samozrejme nemenná.

Séria alebo postupnosť transformácií na jednotlivých RDD vytvára takzvanú RDD rodovú líniu alebo rodokmeň (RDD lineage). Kde je pôvodná RDD reprezentovaná ako rodič a transformáciami vznikajú jej potomkovia.

Vo viac matematickej resp. programátorskej terminológii toto predstavuje logický plán vykonávania jednotlivých transformácií, ktoré sú reprezentované pomocou orientovaného acyklického grafu z angličtiny directed acyclic graph ďalej iba ako DAG.

Formálne je orientovaný graf opísaný pomocou usporiadanej dvojice G = (V, E), kde V predstavuje neprázdnu nekonečnú množinu vrcholov grafu. E v tejto dvojici predstavuje množinu usporiadaných dvojíc typu (x, y), kde $x \neq y$, nazývaných aj ako orientované hrany grafu. Cyklus v grafe alebo kružnica je uzavretá postupnosť prepojených vrcholov viď obr. 12




Obr. 12: Zobrazenie cyklu v orientovanom grafe

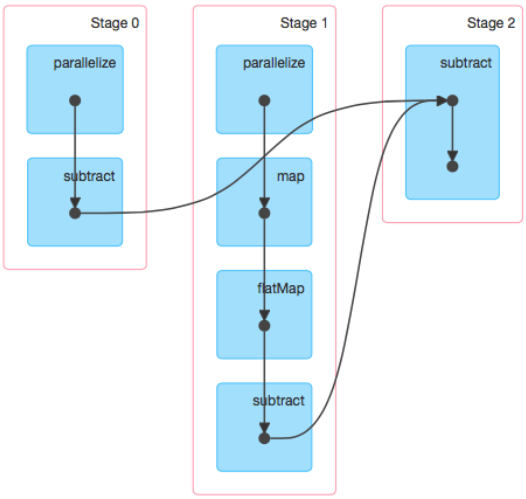
Nakoľko DAG je acyklický tak predstavuje taký graf, ktorý neobsahuje žiadnu kružnicu resp. cyklus. Spark vie takýmto spôsobom reprezentovať postupnosť jednotlivých transformácií, čo koniec koncov prináša viaceré výhody. Medzi ne patrí napríklad dosahovanie odolnosti voči chybám.

Uvažujme nad situáciou, kedy by Spark rozdelil na klaster vykonávanie kódu, ktorý rozdelí do viacerých fáz. Zoberme ako príklad obrázok 13, kedy je vykonávanie rozdelené do troch fáz. Nech Spark rozdelí vykonávanie operácií medzi viaceré uzly klastru.

Počas priebehu vykonávania kódu dôjde k tomu, že jeden z uzlov, ktorý už dokončil jednotlivé operácie z fázy 0 (označme ich O1 a O2) a práve vykonával prvú operáciu z fázy 2 (označme O3), ktorá však zlyhala. Správca klastru zistí, že uzol neodpovedá, a tak sa rozhodne priradiť vykonávanie danej úlohy inému uzlu.

Tým, že sa Spark riadi pomocou DAG však presne vie určiť, že ak prišlo k zlyhaniu v bode O3, tak vie, že danému uzlu musí nechať vykonať aj operácie, na ktorých bola O3 závislá. Tým pádom uzol dostane za úlohu spracovať O1, O2 a následne O3 a týmto mechanizmom sa tým pádom predíde strate dát.

Ďalšou z vlastností transformácií v Apache Spark je ich „lenivosť“(lazy). To znamená,




Obr. 13: Vizualizácia DAG zo Spark Web UI

že pri vykonávaní kódu v Apache Spark sa jednotlivé transformácie nevykonávajú v ten moment, ako sa dostane vykonávacia rutina na daný riadok kódu. Čiže daná transformácia nebude vykonaná okamžite. Spark teda nevykoná danú transformáciu, čo však reálne robí je to, že si postupne buduje svoj plán (DAG), použitím ktorého neskôr rozdelí úlohy uzlom klastra. To, kedy sa vykoná daná transformácia, je popísané nižšie pri akciách. Spark má viacero transformácií, ktoré sa dajú využiť. Patria medzi ne napríklad: *map*, *flatMap*, *filter*, *groupByKey*, *join* a mnohé ďalšie[9]

Transformácie ako také rozdeľujeme do dvoch základných skupín, a to:

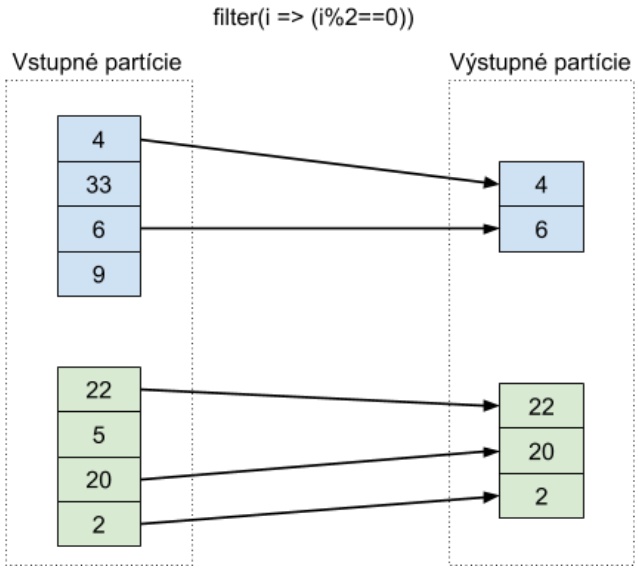
- narrow transformations - „úzke“ transformácie,
- wide transformations - „široké“ transformácie, ďalej ako shuffles.

**Úzka transformácia** je taká, ktorá svojou vstupnou partíciou prispeje k jednej výstupnej partícií[9]. Takouto transformáciou je napríklad *filter*.

Na obrázku 14 môžeme vidieť jednoduchý príklad transformácie *filter*, ktorá má za úlohu vyfiltrovať zo vstupu iba také čísla, ktoré sú párne. To, že je *filter* úzka transformácia možno vidieť na tom, že zo vstupných dvoch partícií každá prispela v rámci výstupu iba do jednej partície.

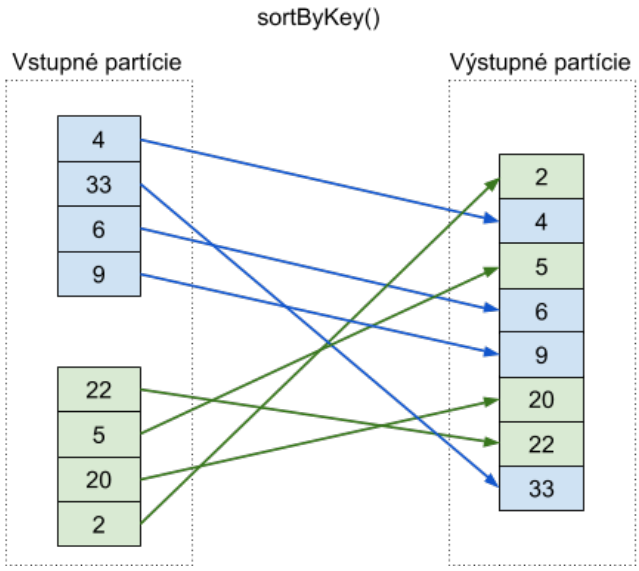
**Shuffles** Shuffles svoj názov dostali práve z toho dôvodu, že na rozdiel od úzkych transformácií prispievajú k výsledku do viacerých partícií, nie iba do jednej. Preto tým pádom dochádza k javu označovanému ako „shuffle“, ktorý zapríčiní výmenu partícií klastra.

[9]Všetky transformácie, ktoré Spark ponúka je možné si pozrieť na nasledujúcom odkaze: `https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations`




Obr. 14: Názorná ukážka transformácie filter

Pri úzkych transformáciách Spark automaticky vykoná operáciu nazývanú pipelining, čo znamená, že ak v dátových rámcoch určíme viac filtrov, všetky sa vykonajú v pamäti. To sa však nedá povedať o shuffle, lebo v takom prípade zapisuje výsledky na disk[9]. Tento typ transformácie sa dá ilustrovať na nasledovnom obrázku 15, kedy sa používa transformácia *sortByKey*.




Obr. 15: Názorná ukážka transformácie sortByKey

K tomu, aby Spark dosiahol požadovaný výsledok zobrazený v tomto príklade, je potrebné usporiadať prvky nie iba v rámci partície nakoľko takáto operácia by nevrátila

usporiadaný celok.

Posledná avšak nie menej dôležitá vec, ktorú treba dodať k shuffle transformáciám je dopad na výkonnostnú stránku. Shuffle je operácia, ktorá môže spôsobovať značné obmedzenia v zmysle výkonu. To práve z toho dôvodu, že zahŕňa diskové operácie, serializáciu dát, ktoré sú zahrnuté v samotnej transformácií a v neposlednom rade aj prenos dát po sieti. Navyše interne musí Spark pri takejto transformácií vygenerovať a rozdeliť súbor úloh v podobe mapovania (pre potrebu usporiadania dát) a súbor úloh pre agregovanie týchto dát[20].

**Akcie** V predchádzajúcej sekcii sme opísali dôležitosť a význam transformácií. Taktiež sme povedali, že riadok kódu, v ktorom je transformácia volaná, neznamená, že jej zmeny budú okamžite aplikované. Spark iba postupne buduje logický transformačný plán. Spúšťadlo tohto plánu vykonávania jednotlivých úkonov predstavuje akcia. Akcia dáva inštrukciu Apache Spark k tomu, aby vykonal sériu transformácií[9].

Akcie sa radia medzi tri skupiny:

- akcie, ktoré zapisujú dáta do výstupných dátových zdrojov,
- akcie, ktoré zobrazujú dáta v konzole,
- akcie, ktoré zapisujú dáta do natívnych objektov príslušného jazyka.

Pre objasnenie toho, ako transformácie spolu s akciami fungujú uvádzame nasledujúci kód 8 v jazyku Scala.

```
val file = sc.textFile("integer-file.txt")
val sortedFile = file.map(_.toInt).sortBy(t=>t)
val outputArray = sortedFile.collect()
outputArray.foreach(println(_))
```

Kód 8: Usporiadanie a výpis čísel zo súboru

V prípade kódu 8 *sc* predstavuje inštanciu triedy *org.apache.spark.SparkContext*, ktorá predstavuje hlavný bod pre sprístupnenie funkcionality Apache Spark[21]. Najprv príkazom *sc.textFile(„integer-file.txt“)* načítame obsah súboru na zadanej ceste, ktorý nám vráti RDD kolekciu reťazcov. Je potrebné povedať, že zvolený textový súbor obsahuje číselné hodnoty, každú z nich na novom riadku súboru a našim cieľom je vypísať usporiadané hodnoty na konzolu.

Následne prevedieme dve transformácie. Prvá z nich je *map*, ktorá namapuje jednotlivé riadky z typu reťazec(*String*) na typ číslo(*Int*). Táto transformácia bola úzkou

transformáciou nakoľko Spark je schopný spraviť mapovanie pre každú partíciu zvlášť bez potreby výmeny údajov medzi jednotlivými partíciami.

Príkaz *map* je však zreťazený s ďalšou transformáciou, a to *sortBy*, ktorá už však predstavuje shuffle, takže Spark musí spraviť všetku potrebnú prácu k tomu, aby mohol usporiadať údaje medzi partíciami. Na záver zavoláme akciu v podobe *collect*, ktorá nám dáta zo Spark-u presunie do premennej outputArray, ktorej obsah necháme vypísať na konzolu.

V tomto príklade transformácie predstavujú operácie *map* a *sortBy*. *Map* má na starosti konverziu z reťazcov, ktoré boli načítané zo súbory, na celočíselné hodnoty. Taktiež je dobré si všimnúť, že map je úzka operácia avšak následne je volaná transformácia *sortBy*, ktorá má na starosti preusporiadanie číselných hodnôt a táto transformácia sa už radí medzi shuffle operácie.

Spark však ani pri jednej z týchto transformácií nespravil nič z dátami, iba si postupne skladal svoj plán, ktorým následne prevedie jednotlivé transformácie. K vykonaniu tohto plánu dôjde až v moment, kedy sa dostane na inštrukciu *collect*. Pri jej volaní sa daný plán vykoná a dáta sú akciou *collect* zozbierané do premennej *outputArray*. Tá má štandardný Scala typ *Array[Int]* a obsahuje Spark-om spracované dáta v podobe usporiadaného poľa čísel.

## 2.4 Streaming v Apache Spark

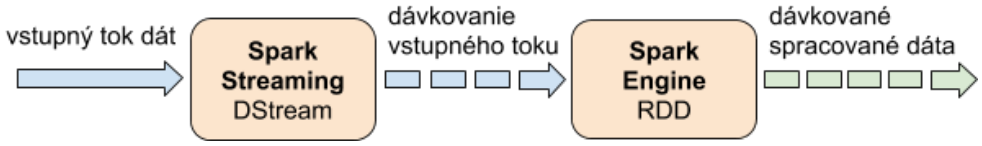
Spark dokáže spracovávať dáta v dvoch módoch. Prvým z nich je dávkový spracovanie (batch processing). Toto je režim v ktorom Spark spracováva dáta, ktoré už boli nejakým spôsobom uložené. Pod týmito dátami je možné si predstaviť napríklad logovacie súbory uložené na serveri alebo údaje, ktoré sú načítané z nejakého databázového systému.

Takéto spracovanie má samozrejme širokú škálu uplatnení nakoľko sa môže používať napríklad periodicky nad nejakými dátami, ktoré je potrebné pravidelne kontrolovať. Taktiež tento režim spracovania môže nájsť svoje uplatnenie ak je potrebné vyhotoviť napríklad štatistiku za nejaké časové obdobie a tak ďalej. Všetky tieto typy použitia však predstavujú spracovanie dát v momente, kedy to užívateľ potrebuje a nie ich spracovanie priebežne ako prichádzajú v čase.

Používateľ však môže potrebovať spracovávanie dát priebežne ako dáta prichádzajú prípadne spracovávanie dát v reálnom čase. Takáto potreba môže nastať ak je potrebné prípadne užitočné získavať štatistiky priebežne prípadne scenáre, kedy je potrebné reagovať na aktuálne správanie užívateľa a tak ďalej. Tento problém rieši Spark využitím svojho druhého módu spracovania údajov a tým je Spark Streaming.

### 2.4.1 Spark Streaming a Spark Structured Streaming

Spark Streaming predstavuje rozšírenie základného rozhrania Apache Spark, ktoré umožňuje škálovateľné, vysoko priepustné spracovanie dátových tokov v reálnom čase[20]. Spark po stránke architektúry využíva pri tomto type spracovávania dát pôvodné koncepty. Toto je zabezpečené tým, že nad tokom dát, ktoré Spark obdrží, používa abstrakciu v podobe diskretizovaného toku z angličtiny Discretized Stream, ktoré sa často označuje skrátenou verziou DStream. DStream predstavuje kontinuálny tok dát, abstrakcia, ktorú ponúka Spark Streaming nad konceptom RDD. Tok dát v tomto prípade Spark nevníma nijak inak ako kolekciu alebo súbor RDD, ktoré daný tok obsahuje. Každý RDD v DStream obsahuje údaje z určitého časového intervalu, ktoré Spark načítal. Spôsob akým operuje nad touto abstrakciou je nasledujúci.




Obr. 16: Zobrazenie spracovania vstupného toku dát pomocou Spark Streaming[20]

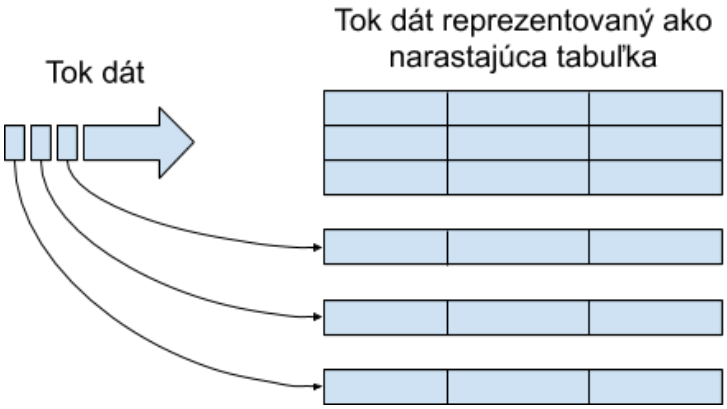
Do Spark Streaming vstupuje tok dát, ktorý reprezentuje ako DStream avšak následne sa tento tok rozdelí do takzvaných mikro dávok z angličtiny micro-batches. S nimi už Spark pracuje ako s klasickými RDD. Navyše, tým, že sa jedná o RDD, Spark dokáže toto spracovanie dát vykonávať paralelne spolu s ostatnými výhodami, ktoré RDD prináša, ktoré sme na túto tému popísali vyššie.

Takéto mikro dávky dokáže Spark následne prerozdeliť a rozposlať jednotlivým uzlom klastru na vykonanie výpočtov. Spôsob akým mikro dávky Spark získava z už spomínaného kontinuálneho toku dát reprezentovaného pomocou DStream je tým, že používa čas dávkovania alebo batch duration[21]. Ten reprezentuje časový úsek, počas ktorého Spark rozdelí pritekajúce dáta do už spomínaných dávok.

Tradičné systémy musia pri zlyhaní nechať prepočítať všetky stratené výpočty a informácie na inom uzle klastru. Iba jeden z týchto uzlov spracováva tieto výpočty a kvôli tomu celý proces musí byť pozastavený až do dokončenia stratených výpočtov. V Apache Spark však v takomto scenári dôjde k tomu, že sa daná úloha, ktorej vykonávanie zlyhalo prerozdelí na jednoduchšie úlohy, ktoré môžu byť následne distribuované k vykonaniu na jednotlivé uzly klastru bez akéhokoľvek ovplyvnenia správnosti výsledkov.

Spark Structured Streaming je ďalším spôsobom, ktorý môže byť použitý pri spracovaní dátových tokov. V tomto type taktiež vystupuje pojem spúšťacieho intervalu, s

ktorým Spark pracuje avšak na rozdiel od Spark Streaming sa v tomto prípade už nestretávam s konceptom dávok.




Obr. 17: Zobrazenie spracovania vstupného toku dát pomocou Spark Structured Streaming[20]

V Spark Structured Streaming nie je dátový tok miesto prerozdelenia na mikro dávky. Je možné tento tok reprezentovať pomocou narastajúcej tabuľky ako je zobrazené na obrázku 17. Nové dáta, ktoré v toku pribudnú sú spracované a následne je na základe nich aktualizovaná tabuľka, v ktorej predstavujú nový riadok.

Ďalším rozdielom, ktorý odlišuje tieto alternatívy, je práve to, že Spark Streaming používa vnútorne RDD pričom Spark Structured Streaming používa dátové sety a rámce. Jednu z výhod, ktoré má Spark Structured Streaming oproti Spark Streaming je možnosť pracovať s časom udalosti.

Čas udalosti predstavuje čas, kedy daná udalosť skutočne nastala. V Spark Streaming je v prípade času pracované s údajom, ktorý predstavuje čas, kedy bola daná udalosť prijatá pomocou Apache Spark. Toto môže avšak nemusí predstavovať problém nakoľko pokiaľ sa daná udalosť oneskorí, tak môže byť zahrnutá do neskoršej dávky ako mala pôvodne patriť. Takýto jav tým pádom môže vniesť do systému nepresnosť prípadne stratu dát.

Spark Structured Streaming však rieši tento problém, a to tým, že zavádza koncept už spomínaného času udalosti alebo aj event time. Tým, že už nepracuje s časom prijatia údajov v Apache Spark, sa môže riešiť pôvodne rozoberaný problém. Ak prídu dáta do systému s oneskorením, pracuje sa s časom udalosti, tým pádom môžu byť tieto dáta korektne spracované, a tým sa zvyšuje presnosť výsledkov. V nasledujúcich sekciách budeme opisovať primárne Spark Structured Streaming avšak podobné výsledky a metódy je možné nájsť aj pre Spark Streaming.

Ďalšie informácie o tom, ako je možné využiť Apache Spark pri spracovaní dátových tokov sme popísali v prílohe A.

## 2.5 Redundancia a podobnosť funkcionality v Apache Spark

Ako bolo možné zistiť z predošlých kapitol, v Apache Spark je viacero konceptov, ktoré sú si veľmi podobné a viacero možností akými možno dosiahnuť rovnaký alebo podobný výsledok. To sa týka predovšetkým RDD, dátových setov, Spark Streaming, Spark Structured Streaming a tak ďalej.

Napríklad RDD a dátové sety. Sú to dve rozdielne aplikačné rozhrania napriek tomu majú veľmi podobnú takmer až zhodnú úlohu v zmysle transformácií a vykonávania akcií nad dátami. Taktiež transformácie, ktoré možno nad nimi prevádzať sú zhodné ako napríklad: *map*, *filter*, *flatMap*, *mapPartitions* a mohli by sme pokračovať ďalej.

Dokonca Spark ponúka pri dátových setoch aj metódu, ktorou možno konvertovať z dátového setu na RDD pomocou metódy *rdd*. Podobnú konverziu je možné spraviť aj pri použití RDD, ktoré je neskôr potrebné previesť na dátový set. V takom prípade je možné použiť metódu *createDataset* na triede *SparkSession*. Aj len v samotnom DataSet API je možné sa rozhodnúť, či bude užívateľ narábať s dátovými setmi pomocou SQL alebo bude volať jednotlivé metódy na zvolenom objekte v jazyku podľa jeho voľby.

Takže niektoré z týchto API majú podobnú funkcionalitu v zmysle výsledku, ktorý používateľ obdrží avšak majú taktiež veľa odlišností v tom, ako interne pracujú a taktiež ako sa s nimi narába pomocou obslužného kódu, ktorý je potrebné napísať. Tvorcovia sa tým pádom snažia dopriať voľnosť v tom ohľade, že je na používateľovi, ktorú API sa rozhodne použiť na splnenie svojej úlohy.

Veľmi podobná situácia nastáva aj ak by sme porovnávali Spark Streaming a Spark Structured Streaming. Toto porovnanie podobne ako v prípade RDD a dátových setov sme bližšie opísali v príslušných kapitolách. Podstatou je však to, že podobne ako v prípade RDD a dátových setov tak aj v prípade Spark Streaming a Spark Structured Streaming existuje množstvo ekvivalentných metód, ktorými možno docieliť rovnaký výsledok.

Pri takomto porovnaní prípadnej redundancie treba však myslieť aj na implementačné detaily, ktoré práve v týchto prípadoch zohrávajú svoju rolu. Dátové sety sú postavené nad Spark SQL, ktorý využíva Catalyst. Catalyst slúži na generovanie optimalizovaných logických a fyzických dopytovacích plánov[18].

Dátové sety okrem toho používajú aj Tungsten. Tungsten predstavuje projekt, ktorý zahŕňa správu pamäte a binárne spracovanie pre odstránenie režijných nákladov pri JVM

objektoch a garbage collection[10] a ďalšie optimalizácie. Tým pádom aj napriek tomu ak sa užívateľ rozhodne použiť RDD alebo aj Spark Streaming, ktorý používa RDD, prichádza o práve spomínané optimalizačné techniky, ktoré môžu, ale nemusia spôsobovať pre daný prípad použitia problém.

## 2.6 Možnosť integrácie kódu modelu v jazyku Python

Jednou z našich úloh bolo preveriť možnosť integrácie kódu modelu jazyku Python. Dôvodom je práve stále rastúca popularita tohto programovacieho jazyku a jeho využiteľnosť v oblasti umelej inteligencie.

Na úvod je potrebné predostrieť modelovú situáciu, ktorá by mohla predstavovať prípad, kedy je potrebné takéto riešenie využiť. Napríklad softvérový projekt, na ktorom pracuje viacero ľudí od programátorov až po ľudí, ktorí sa zaoberajú témou umelej inteligencie.

Takýto projekt môže mať naprogramovanú základnú kódovú infraštruktúru v jazyku Java a jednu z jej častí môže tvoriť Apache Spark. Takáto časť má za úlohu spracovávať dátové toky v systéme pričom tím ľudí zaoberajúci sa umelou inteligenciou by chcel do tohto procesu zapojiť nejaký model, ktorý bol nimi vyhotovený v jazyku Python. Napríklad v takomto prípade môže byť zaujímavé spúšťanie externého Python kódu.

Po vykonaní prieskumu sme našli viacero možností ako by sa táto problematika v jazyku Java za použitia Apache Spark dala riešiť. Uvedieme zoznam možností, ktoré je možné zvážiť pri výbere spôsobu riešenia tohto problému:

- použitie tried Runtime alebo ProcessBuilder,
- Jython,
- použitie webových služieb.

**Runtime alebo ProcessBuilder** Prvou z možností je použitím Java tried *Runtime* alebo *ProcessBuilder*. Každá z nich má mierne odlišné rozhranie a metódy, ktoré poskytuje avšak zaradili sme ich do tej istej skupiny nakoľko pracujú na podobnom princípe. *Runtime* je však z hľadiska verzií, v ktorých vyšla staršia ako samotný *ProcessBuilder*, a preto sme našu pozornosť upriamili viac na túto možnosť.

*ProcessBuilder* sa používa na vytváranie procesov operačného systému. Každá inštancia *ProcessBuilder* riadi kolekciu atribútov procesu pričom narábanie s touto triedou

[10]Garbage collection je v JVM pojem, ktorým sa označuje proces skúmania haldy(pamäte) a zisťovania, ktoré objekty sa používajú a ktoré nie. Taktiež tento proces zahŕňa odstránenie nepoužívaných objektov[22].

je pomerne jednoduché. Pri vytváraní inštancie tejto triedy je potrebné vložiť jednotlivé reťazce, ktoré následne budú spustené ako samostatný proces. Toto spustenie má na starosti metóda *start*, ktorá spustí proces. Výstup skriptu, ktorý môžeme týmto spôsobom vykonať je tak následne možné získať pomocou metódy *getInputStream*.

Týmto spôsobom je tým pádom možné vykonať ľubovoľný Python kód nakoľko sa jedná o spustenie kódu na zariadení s tou verziou Python, ktorá tam je nakonfigurovaná, spolu s knižnicami, ktoré sú k jeho vykonávaniu potrebné a tak ďalej.

Medzi potenciálne nevýhody tohto prístupu však môže patriť napríklad to, že komunikácia medzi jazykmi Java a Python je tým pádom obmedzená na vstupné parametre, ktoré sú zasielané ako vstupy a podobne výstup je limitovaný na reťazce, ktoré budú získané po spustení vykonávania.

To samozrejme limituje prípady použitia a do istej miery ovplyvňuje aj čitateľnosť kódu nakoľko pri zvyšujúcom sa počte vstupov začne byť tento prístup mierne neprehľadný. Taktiež získavanie výstupu môže byť komplikovanejšie. Pri získavaní napríklad jednej návratovej hodnoty je tento prístup v poriadku, veci sa môžu začať komplikovať v prípade, že sa bude za potreby z takéhoto skriptu získať viac návratových hodnôt. V takom prípade je potrebné napísať kód, ktorým sa budú tieto informácie parsovať na strane Java kódu.

Z hľadiska výkonu táto možnosť nepredstavuje najhoršiu voľbu nakoľko nie je za potreby žiadna komunikácia po sieti. Jediné, čo je potrebné je vytvorenie inštancie *ProcessBuilder*, ktorý interne vytvára proces operačného systému[23].

**Jython** Jython je implementácia jazyka Python v jazyku Java. Predstavuje pomerne jednoduchú možnosť ako spustiť Python kód použitím jazyka Java. Avšak po dlhšom skúmaní tejto možnosti sme prišli na značné limitácie tejto možnosti.

Prvým obmedzením tohto riešenia je to, že takýto kód musí byť napísaný vo vnútri Java kódu použitím triedy *PythonInterpreter*. To znamená, že kód modelu, ktorý by bol pôvodne napísaný v jazyku Python vo svojom osobitnom súbore by tak musel byť prepísaný do zdrojového kódu, ktorý bol napísaný v jazyku Java.

Väčším problémom je však to, že veľa knižníc, ktoré sa používajú pri vývoji modelov umelej inteligencie, predstavujú knižnice ako napríklad scikit-learn, ktorá je do značnej miery závislá na knižniciach ako sú numpy a scipy, ktoré majú veľa rozšírení kompilovaných v jazyku C a Fortran a tie nie je možné použiť v Jython[24].

**Webové služby** Ďalšou z možností, ktorú sme zvažovali bolo použitie webových služieb. Táto možnosť sa dá previesť rôznymi spôsobmi. Do úvahy prichádza napríklad použitie

webových frameworkov pre Python ako napríklad Flask či Django hostovaných na vlastnom serveri. Taktiež je možné zvážiť použitie cloudových služieb ako IBM Cloud, Google Cloud Platform a mnohé ďalšie.

Táto možnosť má však taktiež svoje nevýhody. Jednou z nich je zvyšujúca sa komplexita celkovej infraštruktúry nakoľko je potrebné mať server, na ktorom bude táto služba spustená. Tým pribúda ďalší server, ktorý je potrebné udržiavať a monitorovať.

Taktiež je potrebné vziať na ohľad aj to, že môže nastať situácia, kedy je takáto služba nedostupná. V takom prípade nebude mať obslužný program veľa možností, ktoré môže podniknúť k tomu, aby získal svoj potrebný výstup mimo zopakovania pokusu.

**Zhodnotenie** Pokúsili sme sa stručne zhodnotiť potenciálne výhody a nevýhody jednotlivých riešení. Každé z riešení, ktoré sme skúmali má svoje silné a slabé stránky a vo veľkej miere záleží od daného prípadu a jeho požiadaviek, ktoré sú na toto riešenie kladené.

V implementácií aplikácie použijeme prvú možnosť v podobe ProcessBuilder-a, a to práve z toho dôvodu, aby sme demonštrovali jednoduchosť implementácie tohto riešenia s externým Python kódom.

# 3 Identifikovanie obchodne prínosných udalostí

V bankových systémoch sa vyskytuje veľké množstvo udalostí, ktoré sa dejú každú chvíľu. Od záznamov o tom, že si osoba zobrala hypotéku, cez uzavretie poistenia až po výber peňazí z bankomatu. Avšak nie pre každú z týchto udalostí je obchodne prínosné na ňu reagovať v pomerne krátkom časovom úseku. Práve preto je potrebné zvoliť niekoľko udalostí, pri ktorých by bolo z obchodného hľadiska vhodné na tieto udalosti reagovať.

Sústrediť sa môžeme napríklad na také udalosti, ktoré by mohli pomôcť zvýšiť bezpečnosť. Jednou z takýchto udalostí je napríklad prevod peňazí medzi účtami. Prieskum PwC za rok 2020 ukázal, že z vyše 5000 respondentov 47% z nich potvrdilo, že sa v priebehu posledných dvoch rokov stretli s podvodnou činnosťou. Pričom z finančného sektoru bolo 15% podvodov označených za počítačovú kriminalitu[25].

Preto si myslíme, že implementovanie systému na detekciu podvodných platieb spadá práve do kategórie udalostí, na ktoré je obchodne prínosné reagovať pomerne rýchlo. Práve preto, že transakcie by nemali trvať dlhý čas kým budú prevedené a zároveň to je obchodne prínosné pokiaľ bude spoločnosť chránená pred tým, aby sa nemusela vysporiadavať s podvodnými prevodmi.

Takýto typ udalosti dokáže potenciálne ušetriť peniaze avšak existuje pomerne dosť udalostí, ktoré môžu byť prínosné z toho hľadiska, že banka ponúkne svoje služby klientovi, ktoré potrebuje v danom momente. Kľúčové je práve slovné spojenie „v danom momente“. Uvedieme modelovú situáciu. Klient príde k bankomatu, z ktorého si chce vybrať nejaký obnos peňazí. Po zvolení jeho požadovanej sumy mu však systém vyhodnotí, že má nedostatočné prostriedky a zamietne jeho žiadosť o výber. Takáto situácia by však mohla byť vyriešená aj nasledovne. Systém by zistil, že klient nemá dostatok prostriedkov na vykonanie danej operácie avšak po kontrole hodnotenia daného klienta a splnenia nejakých dodatočných podmienok by klientovi v takomto prípade mohlo byť schválené povolené prečerpanie. Tým pádom by systém zaslal túto ponuku klientovi miesto hlášky o zamietnutí výberu.

Úskalie tohto riešenia však môže byť v tom, že na rozdiel od detekcie podvodných platieb, ktorej pravidlá sa tak často nemenia, ponuky, ktoré dostáva klient sa môže banka rozhodnúť meniť, rušiť, pridávať nové a tak ďalej. Inými slovami, predpokladá sa, že ponuky, ktoré budú zadávané biznis oddelením banky sa budú omnoho častejšie meniť. Preto sme sa rozhodli zvoliť aj druhú obchodne prínosnú udalosť, ktorú implementujeme. Implementácia bude využívať “rule-engine” alebo systém, v ktorom bude možné definovať

biznis pravidlá, na základe ktorých sa následne vyvodia dôsledky v kóde.

## 3.1 Technológie

Na vybudovanie takéhoto systému budeme potrebovať viacero technológií. Tie, ktoré sme vybrali a použili sú zobrazené v nasledujúcej tabuľke 1.

| **Názov** | **Verzia** | **Popis** |
|---|---|---|
| Ubuntu | 18.04.4 | Operačný systém |
| Apache Spark | 2.4.5 | Distribuovaný framework pre klastrové výpočty |
| Confluent Platform | 5.4.1 | Enterprise platforma na streamovanie dát |
| Java | 1.8.0_252 | Programovací jazyk |
| IntelliJ IDEA Community | 2019.3.3 | Vývojové prostredie |

Tabuľka 1: Tabuľka použitých technológií

Je potrebné dodať, že použitím Confluent Platform sme na náš systém nainštalovali (a nakonfigurovali) dodatočné veci, ktoré s ním prichádzajú a to konkrétne:

- Apache Kafka,
- Apache Zookeeper,
- kSQL nadstavba.

Nakoľko sme použili operačný systém Ubuntu nepotrebovali sme doinštalovať Python (v našom prípade vo verzií 3.6.9) keďže je už predinštalovaný na operačnom systéme Ubuntu. Java projekt, ktorý sme použili využíva na správu potrebných knižníc Maven. Všetky potrebné knižnice spolu s ich príslušnými verziami, ktoré treba k spusteniu projektu sú definované v súbore pom.xml v koreňovom adresári projektu.

### 3.1.1 Drools

Drools je riešenie pre správu obchodných pravidiel z angličtiny „Business Rules Management System“ ďalej iba ako BRMS. Jeden z dôvodov, prečo je práve na biznis pravidlá vhodné použiť takýto prístup miesto naprogramovania pravidiel priamo do kódu je ten, že biznis pravidlá sa pomerne často menia alebo aspoň môžu meniť. Tým pádom ak by bola takáto logika implementovaná priamo do kódu, bolo by každú zmenu týchto pravidiel potrebné preprogramovať programátorom[26].

Drools používa DRL, čo je skratka pre „Drools Rule Language“. Pomocou neho je možné definovať biznis pravidlá, ktoré sú následne uložené do súboru formátu *drl*. Tento

jazyk má definovanú syntax s vyhradenými kľúčovými slovami, ktoré je potrebné používať pri definovaní jednotlivých pravidiel[27]. Uvažujme jednoduchú triedu produktu, ktorá má ako atribúty svoj typ a zľavu (viď. kód 10).

```
package sk.stuba.fei.bp.entities;

public class Product {
        private String type;
        private int discount;
        public String getType() {
                return type;
        }
        public void setType(String type) {
                this.type = type;
        }
        public int getDiscount() {
                return discount;
        }
        public void setDiscount(int discount) {
                this.discount = discount;
        }
}
```

Kód 10: Java entita produktu

Táto trieda má iba dva atribúty pričom prvý z nich predstavuje typ produktu a druhý predstavuje zľavu, ktorá môže danému produktu prislúchať. Následne je možné definovať súbor pravidiel, ktoré budú na objektoch tejto triedy nastavovať jednotlivé zľavy. Povedzme, že by bolo potrebné nastaviť zľavu 50 eur na produkty, ktoré sú diamantové a na produkty zo zlata zľavu vo výške 30 eur. Takýto súbor pravidiel by použitím Drools mohol vyzerať ako kód 11.

```
package sk.stuba.fei.bp.rule
import sk.stuba.fei.bp.entities.Product
rule "Zlava na diamantove produkty"
        when
                productObject: Product(type=="diamanty")
        then
                productObject.setDiscount(50);
        end
rule "Zlava na produkty zo zlata"
        when
                productObject: Product(type=="zlato")
        then
                productObject.setDiscount(30);
        end
```

Kód 11: Súbor pravidiel pre produkt

Tým pádom sa nám podarilo extrahovať logiku, ktorá by inak bola umiestnená v kóde do súboru pravidiel. Avšak ako je možné vidieť takýto súbor pozostáva z viacerých kľúčových slov, ktorých syntax a použitie je potrebné sa naučiť. Takýto spôsob samozrejme nemusí byť vzdialený programátorovi, avšak môže robiť problém niekomu ako je napríklad biznis analytik.

Dôvod, prečo sme si zvolili Drools je ten, že okrem DRL ponúka možnosť definovať pravidlá prostredníctvom rozhodovacích tabuliek, ktoré možno jednoducho napísať v programoch ako je Excel. Samozrejme, aj tieto rozhodovacie tabuľky majú svoje pravidlá a svoju syntax, ktorú je potrebné zachovať. Avšak pokiaľ je vypracovaná základná šablóna, tak následne nie je za potreby programátor k tomu, aby mohli byť pridané nové pravidlá prípadne upravené alebo odstránené existujúce. Na obrázku 18 je možné vidieť ako by pôvodne definované pravidlá vyzerali v prípade použitia rozhodovacej tabuľky.

Ako je možné vidieť je stále potrebné uviesť objekty, ktorých tried budú v súbore používané (riadok číslo dva s označením „Import“). Taktiež je potrebné odlišovať medzi podmienkami (stĺpce označené ako „CONDITION“) a akciami (stĺpce označené ako „ACTION“). V Drools sa týmito označeniami pomocou podmienok označujú náležitosti, ktoré musia byť splnené a následne v prípade splnených podmienok vykoná akcia.

Podmienok a aj akcií môže byť samozrejme viac. Je nutné poznamenať, že medzi jednotlivými stĺpcami s podmienkami (ak je ich viac) platí logické „or“. Podobne ako je možné mať viacero podmienkových stĺpcov, je možné mať i viac stĺpcov s akciami, ktoré

| | A | B | C |
|---|---|---|---|
| 1 | **RuleSet** | rules | |
| 2 | **Import** | sk.stuba.fei.bp.entities.Product | |
| 3 | **Notes** | Rozhodovacia tabuľka na výpočet zliav | |
| 4 | | | |
| 5 | RuleTable Výpočet zliav | | |
| 6 | NAME | CONDITION | ACTION |
| 7 | | productObject: Product | |
| 8 | | type==$param | productObject.setDiscount($param); |
| 9 | **Názov** | **Typ** | **Nastavenie zľavy** |
| 10 | Zľava na diamantové produkty | "diamanty" | 50 |
| 11 | Zľava na produkty zo zlata | "zlato" | 30 |

Obr. 18: Rozhodovacia tabuľka s pravidlami pre produkty

majú byť vykonané.

Pod samotným typom stĺpca je možné vidieť, že sme uviedli názov objektu spolu s jeho typom resp. triedou. Potom je už možné sprístupňovať jeho atribúty pomocou ich názvov a napríklad ich porovnávať a tak ďalej.

Taktiež je možné okrem atribútov použiť i metódy triedy daného objektu ako je možné vidieť v stĺpci s akciou, kde sa nastavuje zľava pomocou metódy *setDiscount*. Takýto dokument napriek tomu, že taktiež potrebuje dodržiavať istú syntax, bez ktorej nebudú pravidlá korektne vygenerované, môže predstavovať pre ľudí, ktorý sa nezaoberajú programovaním jednoduchší spôsob ako vytvárať, upravovať a mazať pravidlá.

### 3.1.2 Confluent Platform

Confluent Platform je enterprise platforma na streamovanie dát. Vybrali sme ju práve z toho dôvodu, že jej inštaláciou získavame prístup k Apache Kafka, Apache Zookeeper, Streams API a aj ku kSQL. kSQL predstavuje nadstavbu nad Apache Kafka.

Pomocou kSQL je možné použiť SQL dopyty, ktorými možno spravovať dátové toky. Tým pádom nie je potrebné písať kód v jazykoch ako Java alebo Python. Confluent kSQL podporuje širokú škálu operácií ako agregácie, spájania, atď. V našom prípade sme kSQL použili na jednoduché vytváranie KTable, čo je koncept, ktorý bližšie popíšeme v nasledujúcich kapitolách.

Okrem kSQL Confluent Platform prináša taktiež aj Control Center. Control Center je grafické rozhranie, ktoré slúži na základnú prácu s Apache Kafka, kSQL a tiež slúži na monitorovanie.

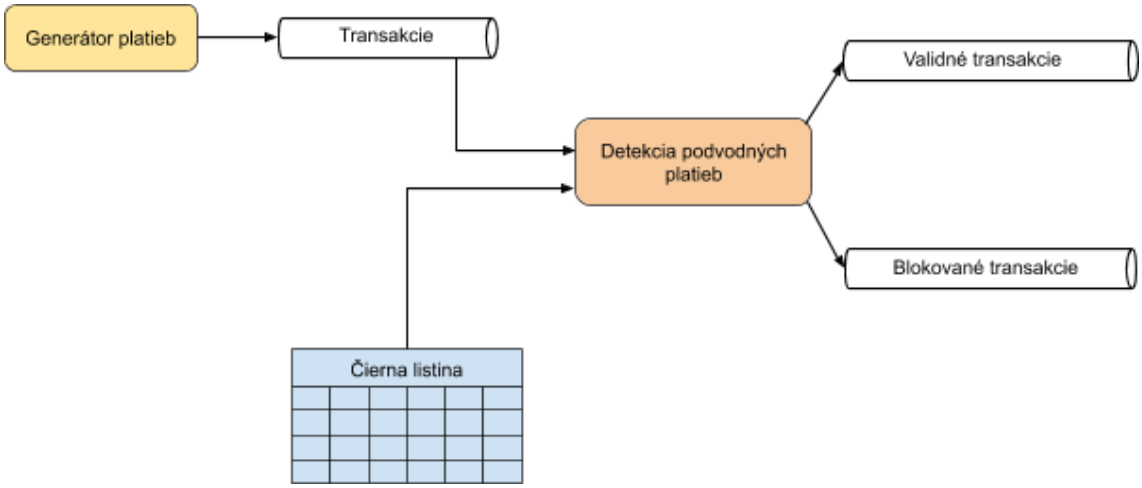
Týmto sme pokryli základné technológie mimo samotného Apache Spark a Apache Kafka, ktoré sme do väčších detailov popísali v predošlých kapitolách.

## 3.2 Návrh architektúry

Pri architektúre nášho riešenia je kľúčové pochopiť to, že niektoré elementy tejto architektúry predstavujú iba náhradu komponentov, ktoré sa snažia simulovať to, ako by takýto systém mohol fungovať pri zapojení do reálneho prostredia. Nakoľko v oboch našich aplikáciách potrebujeme spracovávať toky platieb, či už prevodov medzi účtami alebo kartové transakcie, potrebujeme tak isto aj údaje o klientoch a tak ďalej. Z tohto dôvodu sme sa rozhodli vytvoriť generátory tak ako klientských dát tak i dát o transakciách. Tým pádom sme nemuseli anonymizovať reálne dáta.

Generátor dát by bol však v reálnom použití nahradený za pripojenia k databázam s reálnymi dátami používateľov a časť, ktorá generuje transakcie by bola nahradená pripojením na dátové toky, ktoré prichádzajú z iných systémov.

Prvým znázornením architektúry je práve aplikácia slúžiaca na detekciu podvodných platieb. Táto aplikácia pozostáva z už spomínaných generátorov. Pre jednoduchosť zobrazenia zobrazíme iba ten, ktorý má na starosti generovanie prevodov medzi účtami, generovanie údajov o klientovi, jeho účtoch a kartách (avšak v tomto prípade nie sú karty potrebné nakoľko sa kontrolujú prevody medzi účtami).




Obr. 19: Architektúra aplikácie pre detekciu podozrivých platieb

V našom diagrame predstavujú „Transakcie“ tému, do ktorej sú generované prevody. Čiernu listinu je možné si predstaviť ako tabuľku. V skutočnosti je to však tiež téma (tak s ňou pracuje aj Apache Spark) avšak vytvorili sme nad ňou KTable. KTable predstavuje abstrakciu nad témou a ako sme už pri popisovaní Apache Kafka spomínali, správa v téme je tvorená kľúčom a hodnotou.

KTable robí to, že pre každý kľúč uchováva iba aktuálnu hodnotu. Čo sa hodnoty pre

daný kľúč týka, tak tá predstavuje buď to jednu hodnotu alebo množinu hodnôt vyjadrenú buď to čiarkami oddelené hodnoty, JSON alebo AVRO.

JSON je skratka pre JavaScript Object Notation alebo JavaScript objektový zápis, ktorým možno reprezentovať daný záznam[28]. Avro je jazykovo nezávislá knižnica na serializáciu údajov, ktorá je založená na schémach. Používa schému na vykonanie serializácie a deserializácie. Avro navyše používa formát JSON na špecifikáciu dátovej štruktúry[29].

V prípade funkcionality predstavuje hlavný bod „Detekcia podvodných platieb“, čo je Apache Spark Java aplikácia, ktorá čerpá z tém transakcií a z čiernej listiny. Následne tieto dáta vyhodnotí (skontroluje, či IBAN odosielateľa nie je označený ako blokovaný účet na čiernej listine) a na základe vyhodnotenia prerozdelí transakcie medzi validné a blokované transakcie.

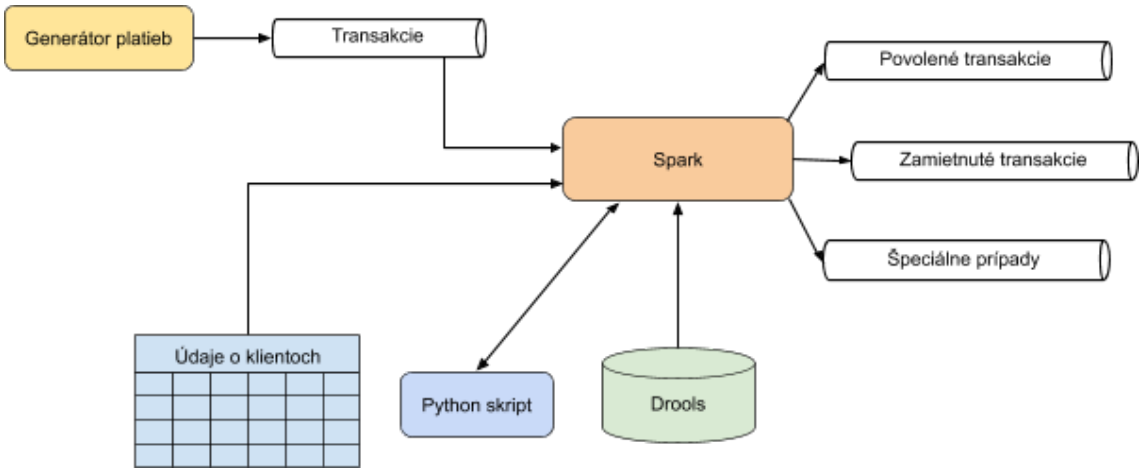
V našej architektúre to pri tomto rozdelení resp. zaslaní transakcií do príslušných tém končí avšak pri implementácií do skutočného systému by mohol ďalší systém odoberať správy z daných tém a vykonávať príslušné akcie. Taktiež naša čierna listina je plnená generátorom náhodne avšak v reálnom systéme by bola plnená dátami z iných systémov, podľa určitých pravidiel, analýz a tak ďalej.

Čo sa týka oboch aplikácií, zdieľajú tú istú dátovú schému (príloha B), ktorej dáta sú rozposielané do jednotlivých tém, nad ktorými sú skonštruované KTable. Treba však brať ohľad na to, že KTable nie sú to isté, čo tabuľky v tradičných relačných databázach. Pri vytváraní KTable sa síce určuje primárny kľúč avšak nie je možné definovať cudzí kľúč. Preto aj v entitno-relačnej schéme nepoužívame cudzie kľúče napriek tomu pri práci s dátami využívame tieto atribúty v rámci operácie *join* teda spájania.

Rozdielnosť medzi dvoma aplikáciami je však v transakčných vetách. V prípade detekcie podozrivých platieb používame transakčné vety pre prevody medzi účtami a v druhom prípade sme použili kartové transakcie.

Je nutné poznamenať, že nad transakciami sme však nevytvárali KTable nakoľko k transakcii z tohto hľadiska pristupujeme ako k udalosti, ktorá nastala a jej stav sa nemení. Nemení sa príjemca, odosielateľ, jej suma a tak ďalej a z toho dôvodu sme ani nepoužili KTable. Napriek tomu sme ju kvôli prehľadu o tom z akých atribútov pozostáva uviedli do diagramu (príloha B). Skripty, ktoré sme použili na vytvorenie KTable je možné nájsť v prílohe (príloha C).

V druhej aplikácii slúžiacej na vyhodnocovanie biznis udalostí sme zvolili architektúru zobrazenú na obrázku 20. Podobne ako v predchádzajúcom prípade je hlavným prvkom Spark aplikácia, ktorá má mimo rozdeľovania transakcií medzi povolené a zamietnuté za úlohu vyfiltrovať transakcie, ktoré sú z biznisového hľadiska relevantné.




Obr. 20: Architektúra aplikácie pre vyhodnotenie biznis udalostí

Toto rozhodovanie je však uskutočňované nielen na základe údajov danej transakcie, ale aj údajov klienta, ktoré Spark čerpá z Apache Kafka. Tie sú následne prehodnotené Python modelom alebo skriptom, ktorý vyprodukuje klientovo skóre. Toto skóre spolu s ostatnými údajmi je následne filtrované pomocou pravidiel, ktoré Spark získa z Drools.

V neposlednom rade na základe získaných údajov pošle transakciu do jedného z troch tém. Transakcie, ktoré nemajú žiaden chybový kód budú zaslané do povolených transakcií. Avšak transakcie, ktoré mali byť pôvodne zapísané do zamietnutých transakcií ešte môžu byť ovplyvnené niektorým z biznis pravidiel a tým pádom sa dostať do tém špeciálnych prípadov. Aj v tomto prípade je dôležité poukázať na to, že napríklad tému špeciálnych prípadov by v reálnom systéme odoberal iný systém, ktorý by následne spracoval tieto informácie - ponúkol klientovi relevantnú ponuku, zaslal mu upozornenie a tak ďalej.

## 3.3 Implementácia a funkcionalita riešenia

Ako sme už spomenuli, jedná sa o Maven projekt, ktorý zachováva štandardnú súborovú štruktúru. Projekt sme si rozdelili na logické celky a podľa toho sme vytvorili aj jednotlivé balíky resp. packages. V priečinku *src/main/java* sa nachádzajú už spomínané dva hlavné zdrojové kódy a to konkrétne:

- *SparkFraudDetection.java* - detekcia podvodných platieb,
- *SparkRuleEngine.java* - detekcia obchodne prínosných udalostí.

Pri architektúre systému sme už popísali entity, s ktorými sa naprieč systémom pracuje. Tie sú reprezentované pomocou dátových tried, ktoré obsahujú svoje atribúty, metódy na ich získanie a nastavenie a taktiež konštruktory. Atribúty majú privátny prístup

a z toho dôvodu sú k nim vytvorené aj metódy na sprístupnenie. Pri transakčných vetách sme použili abstraktnú triedu s názvom *TransactionSentence.java*, z ktorej následne dedí tak ako *TransactionSentenceForTransfer.java* a *TransactionSentenceForCard.java*. Snažili sme sa, aby žiadna z triedy v balíku *entities* neobsahovala žiadnu biznis logiku. Dôvod, prečo sme sa tak rozhodli je ten, že logiku by sme chceli prenechať priamo na kód implementovaný v jednom z dvoch kódov v Apache Spark, prípadne, aby bola veľká časť logiky (pri biznis pravidlách) udržiavaná v pravidlách generovaných pomocou Drools.

V balíku *generator* sa nachádzajú zdrojové kódy, ktoré majú na starosti generovanie jednotlivých objektov. Tieto dáta sú generované náhodne. Každý z generátorov dedí z triedy *Generator*, ktorá má v sebe metódy, ktoré sú vo veľkej miere používané naprieč ostatnými generátormi. *NameGenerator* potrebuje na svoju korektnú činnosť štyri textové súbory lokalizované v priečinku *src/main/resources/names/*.

Tie obsahujú príslušné údaje vždy na novom riadku, tie sú spracované na úvod generátorom mien a ten následne vie na základe týchto údajov generovať náhodné mená.

V balíku *kafka* sa nachádza trieda *KafkaProducer*, v ktorej sme implementovali metódu *procude*. Táto metóda je preťažená, a preto je možné použiť ako vstupné argumenty buď správu na zaslanie a názov témy, do ktorej má byť zaslaná. Alebo v druhom prípade je možné okrem spomenutých argumentov použiť aj kľúč, ktorý bude taktiež zaslaný do príslušnej témy.

```
public RecordMetadata produce(String key, String messageString, String topic) {
    RecordMetadata metadata = null;
    final ProducerRecord<String, String> record = new ProducerRecord<>(topic, key,
        messageString);
    try {
        metadata = producer.send(record).get();
    } catch (ExecutionException | InterruptedException e) {
        System.out.println(e);
    }
    return metadata;
}
```

Kód 13: Metóda pre produkovanie správ do Apache Kafka

Na ukážke kódu 13 je možné vidieť druhú spomínanú metódu. Je potrebné poznamenať, že v konštruktore sa vytvorí objekt *KafkaProducer<String, String> producer*. Ten

slúži k zasielaniu správ, ktorých kľúč je typu *String* a aj samotná správa je formátu *String* - preto aj použitie generiky typu *<String, String>*. V samotnej metóde vytvoríme objekt *record* typu *ProducerRecord<String, String>* pomocou konštruktoru, do ktorého iba delegujeme vstup metódy v príslušnom poradí. Tento objekt následne zašleme metódou *send* na objekte *producer*. Potom je volanie tejto metódy zreťazené metódou *get*, ktorá nám vráti metadáta typu *RecordMedata* o danom zázname.

Je potrebné dodať, že takáto metóda nemusí vždy uspieť práve pri volaní *send*. Preto je potrebné vyriešiť aj prípady, kedy zlyhá toto zasielanie. V našom prípade nakoľko sú tieto metódy používané iba generátorom údajov sme sa rozhodli iba zapísať túto hlášku na konzolu. V produkčných systémov (kde by však nebola metóda používaná náhodným generátorom) by bolo vhodné zalogovať takéto zlyhanie do nejakého logovacieho systému a tak ďalej.

Pred opísaním implementácie konkrétnych aplikácií je ešte potrebné opísať balík *utils*, ktorý obsahuje triedy ako *Configuration* alebo *DataSender*. *Configuration* slúži na spracovanie informácií zo súboru *config.properties*, ktorý sa nachádza v priečinku *src/main/resources/*. V tomto súbore sa nachádzajú konfiguračné údaje a údaje, ktoré sme nechceli vkladať priamo do kódu. Sú tam napríklad cesty k súborom, adresy serverov, názvy tém v Apache Kafka a tak ďalej.

*DataSender* je spojenie generátorov a zasielania správ do Apache Kafka, ktoré sme popísali vyššie. Má za úlohu generovanie údajov klientov, účtov, kariet, naplnenie údajov do čiernej listiny a následné periodické zasielanie transakčných viet do Apache Kafka.

### 3.3.1 Spark Fraud Detection

Táto aplikácia, ako sme už spomínali, má na starosti kontrolu transakcií a ich následné poslanie do jedného z dvoch tém, a to buď do validných alebo blokovaných transakcií. Na úvod je potrebné získať objekt *SparkSession* k čomu je určená *SparkSession*, kde pomocou *builder pattern* vyskladáme objekt s príslušnou konfiguráciou. *Builder pattern* je návrhový vzor, ktorý má poskytovať flexibilné riešenie rôznych problémov pri vytváraní objektov v objektovo orientovanom programovaní[30].

Slúži na nastavenie napríklad mena aplikácie, URL adresy a tak ďalej. V našom prípade sme miesto URL adresy použili hodnotu „local[2]“, ktorá hovorí o tom, že Spark má byť spustený v lokálnom režime. Číslo dva vyjadruje počet vlákien, ktoré má Spark použiť. Dôvod, prečo je odporúčané voliť aspoň číslo dva je ten, že to predstavuje minimálny paralelizmus, pri ktorom je možné aj v lokálnom režime odhaliť problémy a chyby, ktoré by mohli byť v inom prípade nájdené až pri spustení na klastri (kvôli distribuovanému kontextu)[20].

Po základnej konfigurácií nasleduje načítanie čiernej listiny a transakcií. Na to sme vyhotovili dve metódy: *loadBlackList* a *loadTransferStream*. Ako príklad môžeme uviesť *loadTransferStream* a popísať princíp fungovania.

Na tejto metóde (príloha D) je možno vidieť niektoré zo základných vlastností programovania s Apache Spark. Cez *SparkSession* je najskôr zavolaná metóda *readStream*, ktorá vytvorí a vráti nový objekt *DataStreamReader*. Na ňom je zavolaná metóda *format* avšak táto metóda nemá ako návratovú hodnotu *void*, ale vracia upravený objekt triedy *DataStreamReader*. Práve tento prístup umožňuje volať ďalšie a ďalšie metódy tohto objektu a reťaziť ich spôsobom, aký je možné vidieť v prípade načítavania transakčného dátového toku, kde na záver je zavolaná metóda *load*, ktorá vráti objekt typu *Dataset<Row>*. Takýto prístup je často využívaný v Apache Spark.

V tejto metóde (príloha D) je taktiež možné vidieť, že ako možnosti sme uviedli adresu Apache Kafka a názov témy, z ktorej chceme čerpať údaje. Taktiež sme uviedli, že chceme všetky údaje, ktoré už v téme sú pomocou nastavenia *startingOffsets* na *earliest*. Túto možnosť sme zvolili kvôli tomu, aby boli spracované transakcie, ktoré mohli byť v systéme prítomné ešte pred tým, ako bola naša aplikácia zapnutá.

Taktiež sme použili možnosť *enable.auto.commit*, ktorú sme nastavili na *false*. Spolu s Spark *checkpointing* nám to umožní spravovať offsety, ktoré bude odkladať Apache Spark a tým pádom ak dôjde k poruche, aplikácia môže začať čítať správy od offsetu, ktorý bol ako posledný odložený samotným Apache Spark.

Po tomto načítaní je na *transferStream* volaná metóda *selectExpr*, ktorej vstupný parameter je reťazec obsahujúci SQL syntax, ktorou vyberieme z Kafka dátového toku hodnotu (*value*) a následne dáme pomocou funkcie *cast(value as string)* Spark informáciu o tom, že s danou hodnotou má pracovať ako s reťazcom. Tým, že sa jedná o hodnoty oddelené čiarkami tak ich potrebujeme rozdeliť pomocou funkcie *split*. Následne sme jednotlivým hodnotám pridelili názvy pre jednoduchšiu prácu a prehľadnosť. Taktiež sme pomocou funkcie *cast* ešte skonvertovali *transaction_date* a *posting_date* na typ *timestamp* a *currency* na desatinné číslo.

Podobný prístup sme zvolili aj v prípade metódy, ktorá má za úlohu načítať obsah čiernej listiny. Tá obsahuje čísla účtov klientov a príznak, ktorý hovorí o tom, či má byť daný účet blokovaný alebo nie. Z takéhoto dátového toku chceme taktiež získať aj údaj o tom, kedy bol daný príznak nastavený.

Nakoľko môže nastať situácia, kedy klientov účet nie je označený ako blokovaný avšak po nejakom čase príde do daného toku údaj o tom, že daný účet má byť blokovaný. Kafka si však značí k jednotlivým správam v témach aj časovú pečiatku o tom, kedy do danej

témy prišli.

V tomto prípade však na rozdiel od načítavania transakcií sme sa rozhodli uložiť daný dátový tok do pamäte. S dátami sa tým pádom dá narábať pomocou metódy *sql*, ktorú má *SparkSession*. Tú sme využili na spustenie SQL dopytu (kód 15), ktorým sme získali iba aktuálny stav každého účtu, o ktorom sa vedie evidencia. Následne sme využili možnosť Apache Spark v podobne spájania dátových tokov, dátových setov a tak ďalej.

```
Dataset<Row> transfers = transferStream.join(
                blackListCurrent,
                expr("creditor_iban = bl_iban"),
                "inner"
            ).selectExpr("id", "transaction_date", "posting_date", "amount", "type",
                "VC", "SC", "CC", "creditor_iban", "debtor_iban", "blacklisted");
```

Kód 15: Spájanie transakcií s čiernou listinou

V našom prípade sme čiernu listinu spojili s dátovým tokom transakcií, čím sme pre každú transakciu získali údaj o tom, či má byť zamietnutá alebo nie. Následne sme volaním metódy createOutputTransfers získali transformáciami výstupný dátový tok, ktorý je poslaný do Apache Kafka.

Je potrebné zmieniť to, že výstupný dátový tok má formát key, value, topic alebo kľúč, hodnota a téma. To, čo znamená kľúč a hodnota sme už opísali. Apache Spark musí pri zasielaní dát do Apache Kafka povinne definovať aj tému, do ktorej majú byť dáta zaslané. Túto požiadavku je možné naplniť dvoma spôsobmi.

Prvým z nich je pri *writeStream* metóde dodefinovať možnosť topic, ktorej hodnota predstavuje názov témy, do ktorej majú byť dáta zasielané. Táto možnosť však nastaví jednu tému, do ktorej budú poslané všetky správy. Ak by sme týmto spôsobom chceli posielať správy do dvoch tém, museli by sme náš výstupný tok rozdeliť na dva a každý zasielať zvlášť.

Druhou možnosťou je však možnosť vyskladania správy práve v tvare key, value, topic a v takomto prípade Spark zoberie hodnotu pre topic a zašle správy do príslušných tém, tým pádom stačí jedno volanie *writeStream*.

### 3.3.2 Spark Rule Engine

Nasledujúca aplikácia je podobná v niektorých aspektoch tej predošlej. Taktiež je v nej potrebné načítať transakcie (tentokrát však kartové) a taktiež je potrebné odoslať výsledok spracovania do rozdielnych tém (tentokrát však troch a nie dvoch).

Ako sme už v sekcií o architektúre popísali, táto aplikácia načíta transakcie a na základe pravidiel získaných z Drools zistí, či sa v transakciách nenachádza taká, na ktorú by niektoré z týchto pravidiel nebolo možné aplikovať.

V tomto projekte je ešte jeden balík, ktorý nebol opísaný na úvod. Jedná sa o balík drools, ktorý obsahuje iba jednu triedu. Pred tým ako však popíšeme metódy tejto triedy je potrebné ozrejmiť to, čo Drools potrebuje k tomu, aby mohol fungovať v rámci projektu a aké prostriedky potrebuje k vytvoreniu bázy vedomostí alebo pravidiel.

Najprv je dôležité definovať skratku KIE, ktorá označuje v angličtine „Knowledge Is Everything“ čo v slovenčine znamená „Vedomosti sú všetko“, čo je pojem združujúci nástroje ako Drools, jBPM a OptaPlanner. KIE projekt je Maven modul, ktorý však potrebuje navyše ďalšie metadáta k svojmu fungovaniu.

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule">
    <kbase name="AuditKBase" default="true" packages="rules">
        <ksession name="AuditKSession" type="stateless" default="true" />
    </kbase>
</kmodule>
```

Kód 16: XML konfigurácia kmodule.xml

Týmto súborom je kmodule.xml (kód 16), ktorý je umiestnený v priečinku *src/main/resources/META-INF*. Tento súbor je pre Drools konfiguračným súborom, z ktorého získava údaje o zdrojoch znalostí, a to, ako ich má konfigurovať aj spolu s reláciami (sessions), ktoré používa.

Ako je možné z konfigurácie vidieť, použili sme pomerne málo konfigurácie nakoľko v projekte sme využívali iba jednu *kbase*, ktorú sme zvolili za prednastavenú a dali sme jej pokyn brať pravidlá iba z balíku *rules* (štandardne sa pokúsi zostrojiť pravidlá z priečinku *src/main/resources*).

Následne sme ešte definovali jednu *ksession*. Tá slúži na interakciu s Drools pričom záleží na type, ktorý sme v našom prípade zvolili ako stateless, čiže režim, ktorý nezachováva stav. Maven síce dokáže pracovať s prostriedkami KIE avšak neposkytuje validáciu v čase vytvárania (build) projektu. Práve tento problém rieši doplnok (plugin) pre Maven a zároveň aj generuje viaceré triedy čím zrýchľuje následné načítavanie[31].

V tejto aplikácií sa najprv začnú odoberať kartové transakcie z témy z Apache Kafka. Taktiež sa z nej načítajú údaje o klientoch. V tejto aplikácií sme využili schopnosť Apache Spark broadcastovať premenné. Dôvod, prečo je takýto koncept a metóda zavedená v

Apache Spark je práve kvôli distribuovanosti. Nakoľko treba mať na pamäti, že v porovnaní s klasickým kódom v prípade Apache Spark, sú úlohy rozdelené medzi viaceré uzly. Lenže tie tým pádom nezdieľajú tie isté premenné.

Preto ak je v kóde použitá premenná, ktorá nie je zaslaná všetkým uzlom, tak im nebude prístupná. V našom prípade sme potrebovali broadcastovať premennú typu *KieBase*, ktorá obsahuje načítané pravidlá z Drools. Dôvod, prečo sme sa tak rozhodli je ten, že pri vytváraní *KieBase* prichádza k načítavaniu a spracovaniu dát z Drools, čo síce nie je operácia, ktorá by trvala kriticky dlho ak sú tieto pravidlá načítavané raz. Avšak ak by sme nechali načítavať tieto pravidlá pre každú transakciu, ktorých môže byť behom sekundy niekoľko, tak práve táto operácia by mohla patrične zvýšiť čas, ktorý by trvalo dané transakcie spracovať.

Broadcastom tejto premennej k nej tým pádom získame prístup na všetkých uzloch. Po načítaní transakcií a klientských údajov sme následne využili podobne ako pri zisťovaní podvodných platieb možnosť spájania. Po spojení transakcií s klientskými údajmi sme využili transformáciu *map*. Mapovanie sme uskutočnili nasledovne. V prvom rade sme zo spojeného záznamu vyseparovali zvlášť transakciu a zvlášť klienta.

Následne sme údaje o klientovi zaslali do metódy, ktorá slúži na získanie skóre využitím jazyka Python. Túto tému sme opísali v predošlých kapitolách, a preto len poznamenáme, že sme v tomto prípade využili Java triedu *ProcessBuilder*, pomocou ktorej sme spustili Python kód, ktorý vracia klientovo skóre.

Po získaní klientovho skóre nastáva kontrola Drools pravidiel. V rozhodovacej tabuľke, ktorá je umiestnená v priečinku *src/main/resources/rules* sa nachádzajú pravidlá, ktoré budú pri jednotlivých transakciách kontrolované.

Kvôli prehľadnosti sme ich rozdelili na dve časti nakoľko v rozhodovacích tabuľkách môže byť viacero takýchto skupín pravidiel. Pokiaľ sú splnené dané podmienky, tak bude transakcia zaslaná do danej témy. Bude tam zaslaná aj s nastavenou správou, ktorá hovorí o tom, aká situácia nastala, čo je možné spraviť, prípadne akú službu možno klientovi ponúknuť.

Napríklad prvé pravidlo hovorí o klientovi, ktorý sa pokúsil o výber z bankomatu (typ transakcie *„W“*) pričom však nemal dostatok prostriedkov (chybový kód *„INSUFFICIENT_RESOURCES“*). Ak však klient má skóre šesťdesiatosem alebo vyššie a nemá aktivované povolené prečerpanie tak takáto transakcia bude odoslaná do témy *„attention-needed-transactions“*.

Navyše bude zaslaná so správou *„Ponúknuť aktiváciu povoleného prečerpania“*. To by mohlo byť následne vyhodnotené iným systémom a napríklad by klientovi v bankomate

bola zobrazená ponuka o tom, že si môže aktivovať povolené prečerpanie a tým pádom čiastka, ktorú mal v pláne vytiahnuť mu bude vydaná.

Následne napríklad v druhej kategórií pravidiel je prvé pravidlo, ktoré hovorí o tom, že pokiaľ má klient skóre aspoň päťdesiat a jeho momentálny nastavený limit je medzi sto až päťsto eur a jeho platba by mala byť zamietnutá nakoľko prečerpal limit, tak tomu ešte tak nemusí byť. Nakoľko na základe tohto pravidla mu môže byť ponúknuté zvýšenie limitu pre nižšiu kategóriu, o čom môže byť informovaný napríklad SMS správou. Ak sa rozhodne, že by si chcel nechať aktivovať zvýšenie limitu, tak jeho platba môže byť zrealizovaná.

## 3.4 Vlastnosti riešenia

### 3.4.1 Priepustnosť

Priepustnosť je vlastnosť, ktorou sa testuje to, akú záťaž dokáže systém zniesť. Túto vlastnosť sme sa rozhodli názorne demonštrovať na video ukážke, ktorá sa nachádza na elektronickom nosiči pod označením *video-1-priepustnost.mp4*.

Ako je možné vidieť, pri stúpajúcej záťaži v podobe narastajúceho množstva transakcií je Apache Spark stále schopný pracovať bez akýchkoľvek dopadov na funkčnosť systému. V našom prípade sme demonštrovali činnosť produkovania približne 15 transakcií za sekundu avšak tieto čísla môžu byť omnoho vyššie, a to hlavne v prípade klastru.

Veľkosti dát, ktoré dokáže Apache Spark spracovať, sú uvádzané v petabajtoch. Apache Spark v roku 2014 prekonal svetový rekord, kedy v súťaži Daytona GraySort v kategórii utriedenia sto terabajtovej dátovej vzorky zvládol utriedenie na 206 uzlovom klastri za 23 minút[32].

### 3.4.2 Robustnosť a zotaviteľnosť

Robustnosť je vlastnosťou softvéru, ktorý je schopný pokračovať aj v prípade výskytu neočakávaných problémov, ktoré sa môžu počas jeho životného cyklu vyskytnúť[33]. V prípade našich aplikácií sme neimplementovali odchytávanie a logovanie prípadných chýb nakoľko sa jedná o pomerne jednoduchý systém. Avšak tieto aplikácie sú pomerne jednoducho rozšíriteľné o tento mechanizmus. V našom prípade bolo zaujímavejšie to, ako sa bude systém chovať v prípade neobnoviteľných chýb a úplného výpadku systému.

V prípade zotaviteľnosti sme sa sústredili predovšetkým na to, aby naša aplikácia (nakoľko sa jedná o spracovávanie transakcií) bola schopná obnoviť svoju činnosť aj po nečakanom ukončení aplikácie, či už z dôvodu chyby alebo iných faktorov, ktoré vyústia v jej zastavenie. Práve v prípade transakcií je takáto vlastnosť nevyhnutná nakoľko by mohlo dôjsť k nekonzistentným údajom a finančným stratám v prípade, že by sa transakcie

nespracovali, prípadne by boli spracované dva krát a tak ďalej.

V našom riešení sme práve tieto problémy riešili použitím už spomínaných *checkpoints*, ktoré slúžia pre Apache Spark na poznačenie toho, ktorá z transakcií bola spracovaná ako posledná. Slovo „spracovaná“ zohráva v tomto prípade dôležitú rolu nakoľko ak by Apache Spark načítal transakciu z Apache Kafka a aplikácia by ukončila svoju činnosť v ten moment, nemusela by v takom prípade byť spracovaná. Práve preto je pre nás výhodné, že pomocou Apache Spark budeme mať informáciu o tom, ktorá transakcia bola spracovaná ako posledná.

Na demonštráciu sme vyhotovili video ukážku, kde sme predviedli to, ako sa bude práve v takýchto situáciách naša aplikácia správať. Je možné ju nájsť na elektronickom nosiči pod označením *video-2-robustnost-a-zotavitelnost.mp4*.

### 3.4.3 Škálovateľnosť

Nakoľko sme počas implementácie nemali prístup ku klastru, rozhodli sme sa spísať vlastnosti a možnosť škálovania aplikácie v Apache Spark. V predošlých kapitolách sme popísali to, ako Spark dokáže fungovať v rámci klastru. Taktiež sme popísali to, ako Apache Spark rozdeľuje úlohy medzi pracovné uzly klastru.

Práve pomocou navyšovania množstva pracovných uzlov možno škálovať aplikácie v Apache Spark. Spark teoreticky nemá definovanú hodnotu koľko maximálne pracovných uzlov môže byť v rámci klastru avšak najväčší známy klaster ich má osem tisíc[34].

### 3.4.4 Udržiavateľnosť a čitateľnosť

Čo sa udržiavateľnosti a čitateľnosti kódu týka, myslíme si, že takýto kód nie je zrovna náročné pochopiť a udržiavať. Výhodou Apache Spark je napríklad aj to, že údaje, ktoré načíta z dátových tokov je možné previesť na užívateľom definovaný typ použitím triedy *Encoders* prípadne *ExpressionEncoder*, čo koniec koncov môže zlepšiť čitateľnosť kódu.

Výhodou je taktiež možnosť použitia SQL príkazov nad dátami. Toto použitie značne redukuje množstvo kódu hlavne v prípadoch ako napríklad použitie komplikovanejších agregácií, ktoré je pomerne jednoduché spraviť pomocou SQL avšak môžu byť ťažšie čitateľné ak by boli implementované pomocou mapovaní a iných transformácií, ktoré by bolo potrebné napísať v kóde zvoleného programovacieho jazyka.

Celkovo v čitateľnosti kódu pomáhajú práve štruktúrované API Apache Spark v porovnaní s API nižšej úrovne. To bol jeden z dôvodov, prečo sme pri našej implementácií zvolili Spark Structured Streaming a nie Spark Streaming a taktiež dôvod, prečo sme pracovali s dátovými setmi a nie s RDD.

Čo však môže pôsobiť nie zrovna priaznivo na čitateľnosť je práve voľnosť, ktorú

štruktúrované API ponúkajú. Programátor má tak viac možností ako danú úlohu vyriešiť pričom nie je limitovaný na to, aby používal iba SQL. V našej implementácií sme na miestach kde nám to prišlo vhodné použili SQL a v miestach kódu, kde nám prišlo, že danú úlohu bude lepšie vyriešiť použitím jazyku Java sme použili práve tento jazyk.

V prípade väčších aplikácií však môžu byť určené medzi programátormi štandardy a tým pádom môžu používať všetci členovia tímu rovnaké konvencie, čo môže v konečnom dôsledku vyústiť v kód, v ktorom sú problémy riešené tým istým spôsobom a tým zvýšiť čitateľnosť.

Do témy čitateľnosti samozrejme do istej miery vstupuje aj voľba jazyku, v ktorom je užívateľská Apache Spark aplikácia písaná. V našom prípade sme zvolili jazyk Java, ktorý by mal byť najťažšie čitateľný (ako sme už popísali pri porovnaní jazykov). Napriek tomu nám príde kód čitateľný a udržiavateľný. Prepísaním aplikácie do jazyku ako je napríklad Scala by sa však mohla jeho čitateľnosť ešte o niečo zlepšiť.

# Záver

V našej práci sme najskôr predstavili a popísali technológie Apache Kafka a Apache Spark. Pri týchto technológiách sme taktiež uviedli ich využitie, popísali sme ich vlastnosti a koncepty, na ktorých tieto technológie fungujú.

Pri Apache Spark sme taktiež uviedli aj porovnanie jednotlivých jazykov a taktiež sme analyzovali to, ako funguje Apache Spark pri použití týchto jazykov. Okrem iného sme aj preverili možnosť integrovania kódu modelu v jazyku Python do Apache Spark.

Nakoľko sme túto prácu vypracovali v spolupráci so Slovenskou sporiteľňou, tak sme zvolili dva prípady z oblasti bankového sektoru, kde by sa tieto technológie dali aplikovať. Uviedli sme dôvody, prečo má zmysel tieto problémy riešiť a následne sme navrhli riešenie, ktorého technológie a spôsob vypracovania sme opísali a vysvetlili.

Na záver práce sme analyzovali jednotlivé vlastnosti nášho riešenia. V tejto časti sme taktiež pri niektorých vlastnostiach vyhotovili videonahrávky demonštrujúce niektoré zo skúmaných vlastností.

Cieľom našej práce bolo zoznámiť sa so spomínanými technológiami, analyzovať ich vlastnosti a možnosti ich využitia. Následne na základe nadobudnutých poznatkov aplikovať tieto vedomosti a implementovať prototyp systému, ktorý by vedel využiť vlastnosti týchto technológií na obchodne relevantných prípadoch použitia.

# Zoznam použitej literatúry


1. REINSEL, David, GANTZ, John a RYDNING, John. *The Digitization of the World From Edge to Core* [online]. IDC, 2018 [cit. 2019-12-12]. Dostupné z: https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf.
2. *What is Streaming Data?* [online]. Amazon Web Services, Inc. [cit. 2020-01-10]. Dostupné z: https://aws.amazon.com/streaming-data/.
3. *Federal Standard 1037C data stream* [online]. Institute for Telecommunication Sciences, 1996 [cit. 2020-01-09]. Dostupné z: https://www.its.bldrdoc.gov/fs-1037/dir-010/_1451.htm.
4. *Apache Kafka: Introduction* [online]. Apache Software Foundation, 2017 [cit. 2020-01-15]. Dostupné z: https://kafka.apache.org/intro.
5. NARKHEDE, Neha, SHAPIRA, Gwen a PALINO, Todd. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. First. O'Reilly Media, 2017. ISBN 978-1491936160.
6. *Apache Kafka: Documentation* [online]. Apache Software Foundation, 2017 [cit. 2020-01-29]. Dostupné z: https://kafka.apache.org/documentation/.
7. GARG, Nishant. *Learning Apache Kafka*. Second. Packt Publishing, 2015. ISBN 978-1784393090.
8. HALOI, Saurav. *Apache ZooKeeper Essentials*. Packt Publishing, 2015. ISBN 978-1784391324.
9. CHAMBERS, Bill a ZAHARIA, Matei. *Spark: The Definitive Guide: Big Data Processing Made Simple*. First. O'Reilly Media, 2018. ISBN 978-1491912218.
10. HUAI, Yin a ARMBRUST, Michael. *Introducing Window Functions in Spark SQL* [online]. Databricks, 2015 [cit. 2020-04-24]. Dostupné z: https://databricks.com/blog/2015/07/15/introducing-window-functions-in-spark-sql.html.
11. XIN, Reynold, ARMBRUST, Michael a LIU, Davies. *Introducing DataFrames in Apache Spark for Large Scale Data Science* [online]. Databricks, 2015 [cit. 2020-04-01]. Dostupné z: https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html.

12. ZAHARIA, Matei, CHOWDHURY, Mosharaf, DAS, Tathagata, DAVE, Ankur, MA, Justin, MCCAULEY, Murphy, FRANKLIN, Michael, SHENKER, Scott a STOICA, Ion. *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing* [online] [cit. 2020-04-02]. Dostupné z: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf.

13. LUU, Hien. *Beginning Apache Spark 2: with resilient distributed datasets, Spark SQL, structured streaming and Spark machine learning library*. First. Apress, 2018. ISBN 978-1484235782.

14. DEUTER, Margaret, BRADBERY, Jennifer a TURNBULL, Joanna. *Oxford Advanced American Dictionary*. Eight. Oxford Advanced Learner's Dictionary. ISBN 978-0194399661. Dostupné tiež z: https://www.oxfordlearnersdictionaries.com/definition/english/immutable.

15. DEAN, Jeffrey a GHEMAWAT, Sanjay. *MapReduce: Simplified Data Processing on Large Clusters* [online]. 2004 [cit. 2020-04-15]. Dostupné z: https://static.usenix.org/publications/library/proceedings/osdi04/tech/full_papers/dean/dean.pdf.

16. BENGFORT, Benjamin a KIM, Jenny. *Data Analytics with Hadoop*. First. O'Reilly Media, 2016. ISBN 9781491913703.

17. ZAHARIA, Matei, KARAU, Holden, KONWINSKI, Andy a WENDELL, Patrick. *Learning Spark: Lightning-Fast Big Data Analysis*. First. O'Reilly Media, 2015. ISBN 978-1449358624.

18. DAMJI, Jules. *A Tale of Three Apache Spark APIs* [online]. 2016 [cit. 2020-04-23]. Dostupné z: https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html.

19. LASKOWSKI, Jacek. *The Internals of Spark SQL* [online]. Gitbooks [cit. 2020-04-25]. Dostupné z: https://legacy.gitbook.com/book/jaceklaskowski/mastering-spark-sql/details.

20. *Apache Spark Docs* [online]. Apache Software Foundation [cit. 2020-04-28]. Dostupné z: https://spark.apache.org/docs/latest/.

21. *Apache Spark JavaDocs* [online]. Apache Software Foundation [cit. 2020-04-29]. Dostupné z: https://spark.apache.org/docs/2.3.0/api/java/.

22. HUNT, Charlie. *Java Performance*. First. Addison-Wesley Professional. ISBN 978-0137142521.

23. *Java™ Platform: Standard Edition Core Libraries* [online]. 2017 [cit. 2020-04-30]. Dostupné z: https://docs.oracle.com/javase/10/core/process-api1.htm.

24. JUNEAU, Josh, BAKER, Jim, NG, Victor, SOTO, Leo a WIERZBICKI, Frank. *The Definitive Guide to Jython* [online]. Sphinx, 2010 [cit. 2020-04-18]. ISBN 978-1-4302-2528-7. Dostupné z: https://jython.readthedocs.io/en/latest/.

25. *PwC's Global Economic Crime and Fraud Survey 2020* [online]. 2020 [cit. 2020-05-02]. Dostupné z: https://www.pwc.com/gx/en/forensics/gecs-2020/pdf/global-economic-crime-and-fraud-survey-2020.pdf.

26. SALATINO, Mauricio, MAIO DE, Mariano a ALIVERTI, Esteban. *Mastering JBoss Drools 6*. Packt Publishing, 2016. ISBN 978-1783288625.

27. *Designing a decision service using DRL rules* [online]. Red Hat, Inc., 2019 [cit. 2020-05-04]. Dostupné z: https://access.redhat.com/documentation/en-us/red_hat_process_automation_manager/7.2/html/designing_a_decision_service_using_drl_rules/index.

28. *Standard ECMA-404: The JSON Data Interchange Syntax* [online]. Ecma International, 2017 [cit. 2020-05-04]. Dostupné z: https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

29. WHITE, Tom. *Hadoop: The Definitive Guide*. Third. O'Reilly Media, 2015. ISBN 978-1449311520.

30. FREEMAN, Eric, BATES, Bert, SIERRA, Kathy a ROBSON, Elisabeth. *Head First Design Patterns: A Brain-Friendly Guide*. O'Reilly Media, 2014. ISBN 978-0596007126.

31. *Drools Documentation* [online]. Red Hat, Inc., 2017 [cit. 2020-05-10]. Dostupné z: https://docs.drools.org/7.0.0.CR3/drools-docs/html_single/.

32. XIN, Reynold, DEYHIM, Parviz, GHODSI, Ali, MENG, Xiangrui a ZAHARIA, Matei. *GraySort on Apache Spark by Databricks* [online]. 2014 [cit. 2020-05-06]. Dostupné z: http://sortbenchmark.org/ApacheSpark2014.pdf.

33. BIELIKOVÁ, Mária. *Softvérové inžinierstvo - Princípy a manažment*. Vydavateľstvo STU, 2000. ISBN 80-227-1322-8.

34. SANKAR, Krishna. *Fast Data Processing with Spark 2*. Third. Packt Publishing, 2016. ISBN 978-1785889271.

35. KOHAD, Ganesh, GUPTA, Shikha, GANGAKHEDKAR, Trupti, AHIRWAR, Umesh a KUMAR, Ajit. *Concept and techniques of transaction processing of Distributed Database management system* [online]. 2013 [cit. 2020-05-01]. ISSN 2319-9229. Dostupné z: https://www.ijcam.com/paperdownload.php?filename=Concept%20and%20techniques%20of%20transaction%20processing%20of%20Distributed%20Database%20management%20system.pdf.

# Prílohy

# A Spracovanie dátových tokov

V tejto prílohe sme sa rozhodli popísať do väčšieho detailu ako funguje spracovávanie dátových tokov v Apache Spark, ako takéto toky načítať, spracovať a následne akou formou zaslať výstup týchto operácií.

Predtým je však potrebné sa vrátiť k pôvodnej vlastnosti, ktorú sme spomínali, a to konkrétne odolnosť voči chybám. Podmienky, ktoré by na aplikáciu mali byť kladené sú práve odolnosť voči chybám a garancia doručenia dát. Pričom je taktiež potrebné, aby aplikácia v prípade zlyhania bola schopná sa reštartovať presne od miesta, kde skončila, a to bez toho, aby stratila dáta alebo spracovala niektoré dáta duplicitne.

K tomu, aby bolo možné tieto nároky dosiahnuť je potrebné, aby zdroje, z ktorých Spark toky odoberá boli schopné poskytnúť opakované odoberanie dát a taktiež, aby takýto zdroj podporoval "idempotent operiations" čiže operácie, ktorých zopakovaním na rovnakom vstupe vyprodukujú rovnaký výstup. Toto boli náležitosti, ktoré musia spĺňať zdroje tokov avšak to samozrejme nestačí pre obnoviteľnosť po zlyhaní.

Spôsob, ktorým Spark rieši obnoviteľnosť po páde pri práci s dátovými tokmi, je práve používanie kontrolných bodov označovaných ako "checkpoints" a taktiež použitím "write-ahead logs" čo by sa dalo voľne preložiť ako "pred zaznamenávanie zápisov" avšak často sa používa skratka WAL. WAL predstavuje techniku, ktorá zabezpečuje atomicitu (atomicity) a taktiež trvanlivosť (durability) ktoré predstavujú dve zo štyroch základných ACID[11] vlastností pri databázových transakciách. Zmeny sa najskôr zaznamenajú do denníka (log) a ten sa musí zapísať do stabilného úložiska predtým ako sa zmeny aplikujú.

Čo sa týka checkpoint procesu, ten predstavuje proces, pri ktorom sa ukladajú metadáta do užívateľom zvoleného adresára. Vďaka týmto údajom vie tým pádom Spark obnoviť svoj chod v mieste, kde skončil pred pádom a tým pádom nepríde k strate údajov[20].

Čo sa týka dátových zdrojov, tie pozostávajú z niekoľkých typov, ktoré sú už v Apache Spark natívne podporované. Medzi vstupné dátové zdroje v Spark Structured Streaming partia:

- súborový vstupný zdroj,

[11] ACID predstavuje začiatočné písmená slov atomicity, consistency, isolation, durability alebo atomicita, konzistencia, izolácia a trvanlivosť. Toto sú vlastnosti, ktoré zaručujú spoľahlivosť databázových transakcií[35].

- Kafka vstupný zdroj,
- soketový vstupný zdroj,
- rate (rýchlostný) vstupný zdroj.

**Súborový zdroj** číta súbory z priečinku a následne ich zasiela ako dátový tok. Tento typ zdroja podporuje rôzne formáty súborov ako napríklad textový formát, csv, json, orc a parquet.

**Kafka zdroj** poskytuje možnosť čítania dátových tokov z Apache Kafka. Apache Spark ponúka možnosť konfigurácie tohto dátového zdroja, kde užívateľ dokáže nastavovať odkiaľ budú toky načítavané, ako sa budú dáta serializovať a tak ďalej.

Je nutné poznamenať, že posledné dva z vymenovaných zdrojov sú určené na testovacie účely. To znamená, že tieto zdroje nie sú odporúčané na použitie v produkčnom prostredí.

**Soketový zdroj** slúži na čítanie dát v kódovaní *UTF8* a nie je pri ňom zaručená odolnosť voči chybám.

**Rate zdroj** slúži na generovanie dát pri určenom počte riadkov za sekundu, pričom tieto vygenerované dáta obsahujú údaj o čase vygenerovania a samotné dáta. Takýto typ zdroja sa dá použiť napríklad na testovanie výkonu aplikácie.

Po opísaní vstupných zdrojov dátových tokov sú na rade výstupné zdroje:

- súborový výstup,
- Kafka výstup,
- foreach výstup,
- konzolový výstup,
- pamäťový výstup.

**Súborový výstup** ukladá údaje do zvoleného priečinka, pričom formáty pre uloženie súboru sú tie isté ako v prípade načítania súborov.

**Kafka výstup** funguje podobne ako jeho vstupná varianta s tým, že spracované dáta sa nie načítavajú, ale ukladajú do príslušných tém.

Následne je možnosť **foreach výstupu**, ktorým sa dá definovať vlastný spôsob zapisovania dát ako výstupu. Je to možné preťažením niektorých metód triedy *ForeachWriter*.

```
stream.writeStream().foreach(
  new ForeachWriter<Integer> {
    @Override public boolean open(long partitionId, long version) {
      // Otvorenie spojenia
    }
    @Override public void process(Integer record) {
      // Zapisanie cisla
    }
    @Override public void close(Throwable errorOrNull) {
      // Zatvorenie spojenia
    }
  }
).start();
```

Kód 9: Ukážka preťaženia metód ForeachWriter triedy

Ako je možné vidieť na ukážke kódu 9 preťažením zobrazených metód je možné implementovať vlastný výstupný mechanizmus pre zvolený dátový tok.

V neposlednom rade je ešte možnosť **využiť konzolový** a **pamäťový výstup**. Prvý z nich slúži na vypísanie dát do konzoly a druhý slúži na uloženie dát vo forme pamäťovej tabuľky, s ktorou sa dá následne pracovať, vypísať ju a tak ďalej.

**Výstupné režimy** K výstupom sa viažu taktiež aj módy, v ktorých je možné tieto dáta na výstup používať. Apache Spark ponúka tri takéto módy, a to konkrétne:

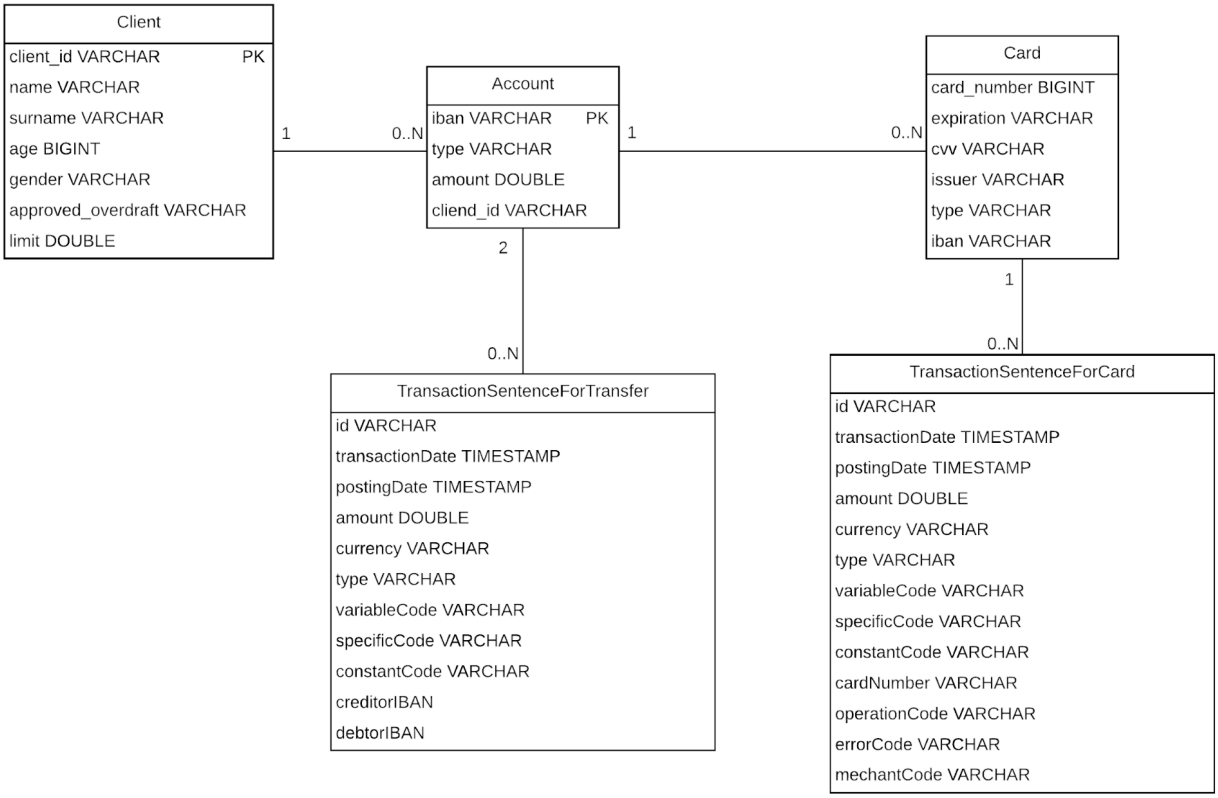
- append mode (režim pripojenia),
- complete mode (kompletný režim),
- update mode (režim aktualizácie).

**Append mode** je predvolený režim, pri ktorom sa pridávajú iba nové riadky do výslednej tabuľky nakoľko všetko do posledného vykonávania bude odoslané do výstupu. Je taktiež potrebné poznamenať, že takýto mód je možné použiť iba v takých prípadoch, pri ktorých sa pridané riadky nebudú meniť. Tento mód možno použiť napríklad pri *select*, *where*, *map*, *flatMap*, *filter*, *join* a tak ďalej.

**Complete mode**, ako z názvu módu vyplýva, predstavuje mód, kde všetky riadky budú figurovať vo výstupe avšak takýto mód je možné použiť iba za predpokladu, že v užívateľskom dopyte alebo kóde je použitá agregácia.

**Update mode** je najnovší z doteraz spomenutých módov výstupu a spočíta v tom, že vo výstupe budú figurovať iba tie záznamy, ktoré boli od posledného výstupu zmenené inak nebudú zahrnuté v nasledujúcom výstupe[20].

# B Entitno-relačný model




Obr. B.1: Entitno-relačný model pre obe aplikácie

# C KTable skripty

```
CREATE TABLE blacklist_table
  (iban VARCHAR,
   blacklisted  VARCHAR)
  WITH (KAFKA_TOPIC = 'blacklist-topic',
        VALUE_FORMAT='DELIMITED',
        KEY = 'iban');
CREATE TABLE account_table
  (iban VARCHAR,
   type VARCHAR,
   amount DOUBLE,
   client_id  VARCHAR)
  WITH (KAFKA_TOPIC = 'account-topic',
        VALUE_FORMAT='DELIMITED',
        KEY = 'iban');
CREATE TABLE card_table
  (card_number BIGINT,
   expiration  VARCHAR,
   cvv VARCHAR,
   issuer  VARCHAR,
   type VARCHAR,
   iban VARCHAR)
  WITH (KAFKA_TOPIC = 'card-topic',
        VALUE_FORMAT='DELIMITED',
        KEY = 'card_number');
CREATE TABLE client_table
  (client_id  VARCHAR,
   name VARCHAR,
   surname VARCHAR,
   age BIGINT,
   gender VARCHAR,
   approved_overdraft VARCHAR,
   "LIMIT" DOUBLE)
  WITH (KAFKA_TOPIC = 'client-topic',
        VALUE_FORMAT='DELIMITED',
        KEY = 'client_id');
```

Kód 12: Skripty pre vytvorenie jednotlivých KTable

# D Metóda načítania dátového toku

```
public static Dataset<Row> loadTransferStream(SparkSession spark, String kafkaEndpoint,
    String kafkaTopic) {
    Dataset<Row> transferStream = spark.readStream()
            .format("kafka")
            .option("kafka.bootstrap.servers", kafkaEndpoint)
            .option("subscribe", kafkaTopic)
            .option("startingOffsets", "earliest")
            .option("enable.auto.commit", "false")
            .load();

    return transferStream.selectExpr("split(cast(value as string),',') as values")
            .selectExpr(
                    "values[0] as id",
                    "to_timestamp(values[1], 'yyyy-MM-dd HH:mm') as transaction_date",
                    "to_timestamp(values[2], 'yyyy-MM-dd HH:mm') as posting_date",
                    "cast(values[3] as double) as amount",
                    "values[4] as currency",
                    "values[5] as type",
                    "values[6] as VC",
                    "values[7] as SC",
                    "values[8] as CC",
                    "values[9] as creditor_iban",
                    "values[10] as debtor_iban"
            );
}
```

Kód 14: Načítavanie dátového toku transakcií do aplikácie

# E Štruktúra elektronického nosiča

*/apache-spark-app*

· Obsahuje Maven projekt opisovaný v práci

*/apache-spark-app/pom.xml*

· XML súbor s potrebnými knižnicami, ktoré boli využité

**/apache-spark-app/src/main/resources**

· Priečinok obsahujúci potrebné súbory k behu aplikácie

*/config.properties*

· konfiguračný súbor

*/client-rating-script.py*

· Python kód pre hodnotenie klientov

*rules/rules.xls*

· Rozhodovacia tabuľka s biznis pravidlami pre Drools

*/META-INF/kmodule.xml*

· Konfiguračný súbor pre Drools

**/apache-spark-app/src/main/resources/names**

· Priečinok s textovými súbormi, ktoré obsahujú mená a priezviská pre generátor

**/apache-spark-app/src/java**

· Obsahuje zdrojový kód aplikácie

*/video-1.mp4*

· Videonahrávka 1

*/video-2.mp4*

· Videonahrávka 2