ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta stavební

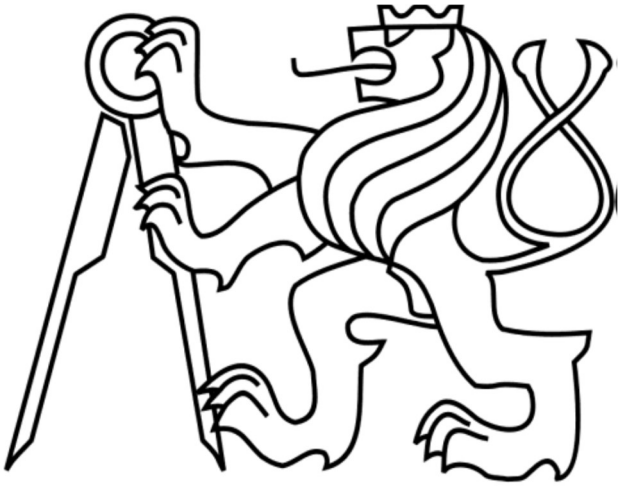
Thákurova 7, 166 29 Praha 6

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FAKULTA STAVEBNÍ

STUDIJNÍ PROGRAM GEOMATIKA

OBOR GEODÉZIE A KARTOGRAFIE



DIPLOMOVÁ PRÁCE

Využití konceptu Big data v oblasti geodézie a kartografie
Usage of Big Data in Geodesy and Cartography

Vedoucí práce: Ing. Jan Pytel, Ph.D.
Katedra geomatiky

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta stavební

Thákurova 7, 166 29 Praha 6

Květen 2022 Martin Vajner

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: Vajner Jméno: Martin Osobní číslo: 473434

Zadávající katedra: 11155 - Katedra geomatiky

Studijní program: magisterský navazující na bakalářský

Studijní obor/specializace: 36T0GKGM - Geomatika

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce: Využití konceptu Big data v oblasti Geodézie a Kartografie

Název diplomové práce anglicky: Usage of Big data in Geodesy and Cartography

Pokyny pro vypracování:
Cílem diplomové práce je analýza využití Big data v oblasti geodézie a kartografie. Neb koncept uložení Big data je všeobecně známý (Apache Hadoop, …) bude diplomová práce zaměřena na využití analytických nástrojů jako je sparc.apache.org. Součástí práce bude praktická ukázka využití analytického nástroje pro analýzu a práci s přicházejícími daty v reálném čase.

Seznam doporučené literatury:
spark.apache.org
Apache Hadoop.apache.org
Bill Chambers - Spark The Definitive Guide: Big data Processing Made Simple

Jméno vedoucího diplomové práce: Jan Pytel

Datum zadání diplomové práce: 4.2.2022 Termín odevzdání DP v IS KOS: 15. 5. 2022
Údaj uveďte v souladu s datem v časovém plánu příslušného ak. roku

Podpis vedoucího práce Podpis vedoucího katedry

III. PŘEVZETÍ ZADÁNÍ

Beru na vědomí, že jsem povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je nutné uvést v diplomové práci a při citování postupovat v souladu s metodickou příručkou ČVUT „Jak psát vysokoškolské závěrečné práce“ a metodickým pokynem ČVUT „O dodržování etických principů při přípravě vysokoškolských závěrečných prací“.

Datum převzetí zadání Podpis studenta(ky)

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta stavební

Thákurova 7, 166 29 Praha 6

Abstrakt

Cílem diplomové práce je analýza využití nástrojů pro zpracování Big data v oblasti geodézie a kartografie. Práce je zaměřena na využití analytického nástroje Apache Spark (sparc.apache.org) používaného spolu s programovacím jazykem Python za použití modulu PySpark. Práce obsahuje teoretické seznámení s konceptem Big data, datovou analýzou jako vědním oborem a nástroji použitými pro samotné zpracování dat. Součástí práce jsou praktické ukázky využití analytického nástroje pro analýzu a práci s daty.

Klíčová slova

Big data, Apache Spark, Python, PySpark, geodézie, datová analýza

Abstract

The aim of the thesis is to analyze the use of tools for Big data processing in the field of geodesy and cartography. The work is focused on using Apache Spark analytic engine (sparc.apache.org) in combination with Python programming language with the support of PySpark module. It contains a theoretical introduction to the concept of Big data, data analytics as a scientific field, and the tools used for data analysis itself. Part of the thesis are practical examples of using analytic engines for data analysis.

Keywords

Big data, Apache Spark, Python, PySpark, geodesy, data analytics

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta stavební

Thákurova 7, 166 29 Praha 6

Prohlášení:

Prohlašuji, že jsem tuto diplomovou práci s názvem Využití konceptu Big data v oblasti geodézie a kartografie vypracoval samostatně, pouze za odborného vedení vedoucího diplomové práce Ing. Jana Pytla, Ph.D. Veškeré podklady, které jsem použil pro tuto bakalářskou práci, jsou uvedeny v seznamu použité literatury.

V Praze dne: 8. 5. 2022 Bc. Martin Vajner

Thákurova 7, 166 29 Praha 6

Poděkování

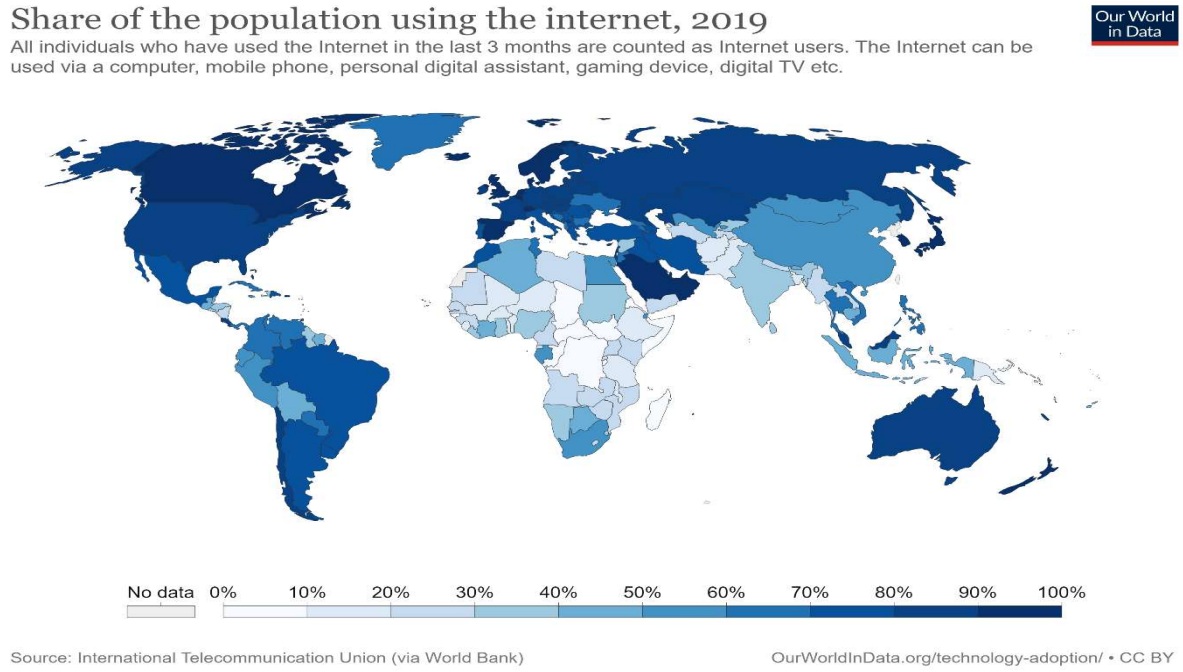
Tímto bych rád poděkoval vedoucímu diplomové práce, Ing. Janu Pytlovi, Ph.D., za poskytnutí námětu pro bakalářskou práci a cenné rady a pomoc při jejím zpracování. Dále bych chtěl poděkovat panu Ing. Zdeňku Vyskočilovi, Ph.D., za poskytnutí dat k této diplomové práci.

Thákurova 7, 166 29 Praha 6

Obsah

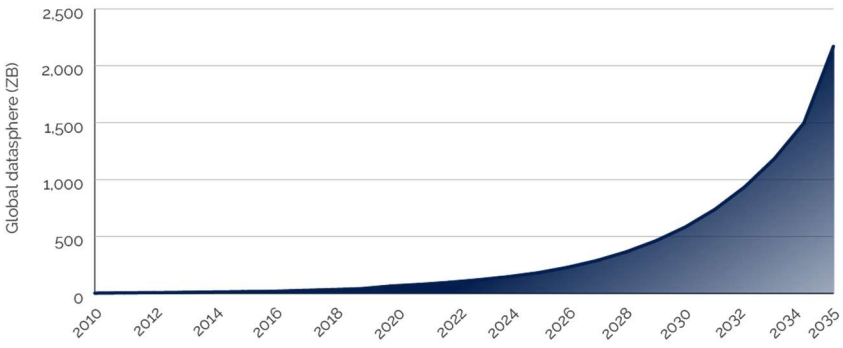
ÚVOD

Lidstvo v posledních letech prošlo neskutečně rychlým vývojem a s tímto vývojem přichází i dramatický nárůst dat. Internet sám je toho důkazem. Internet je globální počítačová síť, která propojuje obrovské množství strojů a umožňuje přenášet informace z jednoho stroje do druhého. V roce 2020 byl počet uživatelů internetu roven hodnotě 4,8 bilionu, což znamená, že skoro 60% populace zanechalo stopu v digitálním světě. Dnes již internet použije denně 5 bilionů lidí. S takovým nárůstem uživatelů dochází i k růstu dat uložených v síti Internet.



Mapa 1 Procento obyvatel užívajících internet (zdroj: ourworldindata.org/internet)

V roce 2013 vydalo IBM studii, podle které bylo 90 % veškerých dat na světě vytvořeno během posledních dvou let. Podle Digital Economy Compass jen v roce 2018 vygeneroval svět 33 zettabajtů[1] dat (1). Co je však zdrojem těchto dat? V podstatě jakákoliv lidská činnost, která je kdekoliv zaznamenána. Data vytváří každé kliknutí myší nebo dotyk v telefonu, interakce na sociálních sítích, chůze po ulici s telefonem v kapse, který sleduje vaši polohu pomocí senzorů GPS; od záznamů z kamer, výpisů z bankovních účtů až po geodetická měření nebo pořizování satelitních snímků. Vše vytváří data a tato data zaplňují Internet rychlým tempem.



Graf 1 Predikce počtu dat (zdroj: holon.investments)

[1] 1 zettabajt = 10^{21} bajtů

Nutně se tedy nabízí otázka, jak efektivně využít takové množstv dat? Firmy využívají data o aktivitě na internetu a umožňují nabízet personalizovaný obsahu uživatelům. Vytváří se softwary na automatizaci a strojové učení pro zpracovávání velkého množství dat. Ale i přes to všechno veškerá data, která člověk vytvoří, zpracovat nelze. Při současném tempu růstu množství dat je to nemožné. A existují nějaká podobná data o takovém objemu v geodézii či kartografii?

Pokud vezmeme v úvahu jeden den měření totální stanicí, o žádné velké množství dat se z pohledu celého světa nejedná. V tomto kontextu je sto naměřených bodů pouze zrnko písku v poušti. Avšak co když budou naměřených bodů desetitisíce, nebo miliony? Na zpracování velkého množství dat již existují analytická řešení a nástroje. Například Python a Apache Spark jsou nejvíce používanými prostředky v analytickém průmyslu, a proto se tato práce zaměřuje na zpracování dat s jejich pomocí (více o zmíněných nástrojích v dalších kapitolách). Ale lze je vůbec efektivně použít pro data prostorová? A existuje vůbec název pro tento fenomén velkého množství dat?

Na všechny tyto otázky se pokusím ve své práci najít odpověď. První kapitola je věnována popisu velkých dat a s nimi asociovaných termínů a datům v geodézii a kartografii. Ve druhé kapitole je přiblížen samotný proces analýzy dat, její popis, kroky a typy. Třetí kapitola se věnuje samotným nástrojům na zpracování dat i jejich předchůdcům, jejich součástem a základům fungování. Ve čtvrté kapitole jsou podrobně popsány vytvořené základní geodetické aplikace pro práci s daty a kapitola pátá patří zhodnocení efektivity zpracování dat pomocí nástrojů Python a Apache Spark.

1. BIG DATA

Tato kapitola obsahuje seznámení s termíny používanými v souvislosti s velkým množstvím dat, jejich popis, typy, zdroje i jejich výhody a nevýhody. Druhá část kapitoly je věnována Big datům v oblasti geodézie a kartografie.

1.1. Big data obecně

1.1.1. Popis

Pojem Big data (v doslovném překladu „Velká data“) se užívá v souvislosti s velkými různorodými soubory informací, které neustále a nestále narůstají. Zahrnuje objem informací, rychlost, jakou jsou vytvářeny a shromažďovány, a rozmanitost nebo rozsah pokrytí. Jejich původ je různorodý, stejně tak jejich formát.

- Big data tedy označují velké množství různorodých informací, které získáváme ve stále větších počtech se stále se zvyšující rychlostí.
- Big data lze strukturovat (často jsou informace číselně vyjádřené, snadno formátovatelné a uchovávatelné), nebo je lze ponechat nestrukturovaná (méně kvantifikovatelná).
- Big data jsou užívána každý den, v podstatě v každém odvětví, a data z nich extrahovaná jsou cennými poznatky. Problémem je ovšem manipulace s nimi, jelikož obsahují zpravidla velké množství tzv. informačního šumu.

1.1.2. Princip využití

Big data se rozdělují do dvou kategorií: strukturovaná a nestrukturovaná. Strukturovaná data obsahují informace, které již prošly filtrem a byly setříděny a roztříděny do databází a tabulek.

Většinou jsou číselné povahy. Data nestrukturovaná jsou, jak již název napovídá, data neorganizovaná, neroztříděná do žádných tabulek a nemají tedy žádný daný model nebo formát.

Big data jsou většinou ukládána v databázích a analyzována pomocí k tomu určených softwarových řešení, vyvinutých k operacím s velkými komplexními soubory dat. Velká data lze shromažďovat z veřejně sdílených komentářů na sociálních sítích a webových stránkách, dobrovolně shromážděných z osobní elektroniky a z aplikací prostřednictvím dotazníků, nákupů produktů a elektronických odbavení. Přítomnost senzorů a dalších vstupů v chytrých zařízeních umožňuje shromažďovat data v širokém spektru situací a okolností.

Datoví analytici sledují vztah mezi různými typy dat, jako jsou demografické údaje a historie nákupů, aby zjistili, zda existuje korelace. Taková hodnocení může provádět interně nebo externě třetí strana, která se zaměřuje na zpracování velkých dat do použitelných formátů. Firmy často využívají k hodnocení velkých dat odborníky, kteří je přemění na použitelné informace.

1.1.3. Problematika efektivního využití

Big data v dnešní době vznikají v podstatě automaticky. Například každá aktivita uživatelů na internetu je zaznamenána a uložena. Každá velká firma má dostatek informací o komkoliv a čemkoliv, ale problémem je jejich využití a zpracování. Ne každý dokáže data správně agregovat a následně zpracovat tak, aby z nich vytěžil užitečnou informaci.

Nárůst velikosti dostupných dat je tedy výhodou i nevýhodou zároveň. V zásadě více informací o např. zákaznících (aktuálních či potencionálních) umožňuje firmám lépe cílit produkty a marketing s cílem zvýšit úroveň spokojenosti a povědomí o firmě a jejích výrobcích. Firmy s přístupem k Big data tedy mají možnost provádět hlubší analýzu např. trhu a zvyšovat tak svou konkurenceschopnost.

Možnost hlubší analýzy má sice pozitivní dopad, ale nutnost filtrovat informační šum snižuje efektivitu tohoto procesu. U velkého množství dat je potřeba velice důkladně rozlišovat, která informace je přínosná a která ne. Data mohou být rovněž chybně zaznamenána a jejich detekce není vždy jednoduchou záležitostí. Toto rozhodnutí je tedy v oblasti Big data rozhodujícím faktorem, zda výsledek analýzy bude blíže nebo dál skutečnosti.

Kromě toho může povaha a formát údajů vyžadovat zvláštní zacházení, než se s nimi začne pracovat. Strukturovaná data, skládající se z číselných hodnot, lze snadno ukládat a třídit. Nestrukturovaná data, jako jsou e-maily, videa a textové dokumenty, mohou vyžadovat použití sofistikovanějších technik, než se stanou užitečnými.

1.1.4. Termín Big data a asociované termíny

Termín Big data byl oficiálně zařazen do Oxfordského slovníku v roce 2013. Avšak první osobou, která jej užila v moderní významu, byl Roger Mougalas již v roce 2005. Právě on představil pojem Big data jako velký soubor dat, který téměř nelze zpracovat pomocí tradičních nástrojů.

Společně s Big data jsou často zmiňována tzv. metadata. Metadata jsou ve zjednodušeném smyslu data/údaje o datech samotných. Neobsahují však nic z obsahu dat, o kterých podávají informace.

1.1.5. Znaky

U Big dat nalezneme většinou uvedené tři základní znaky: Volume, Velocity, Veracity. V některých článcích je jich uvedeno i více, a proto jsem se rozhodl uvést jeden znak navíc k již zmíněným.

- **Volume**
  Objem dat. Jejich velikost.
- **Velocity**
  Rychlost, jakou jsou data přenášena, generována, vyrobena, vytvořena nebo obnovena.
- **Variety**
  Formát dat. Strukturovaná a většinou nestrukturovaná data: zvuk, obrázek, video soubory a další textové formáty, log soubory, data z měření, data ze strojů – senzorů atd.
- **Veracity**
  Pravdivost, důvěra, spolehlivost. Spolehlivost zdroje dat, jeho kontext a jaký má smysl pro analýzu, na něm založenou, a její výsledky.

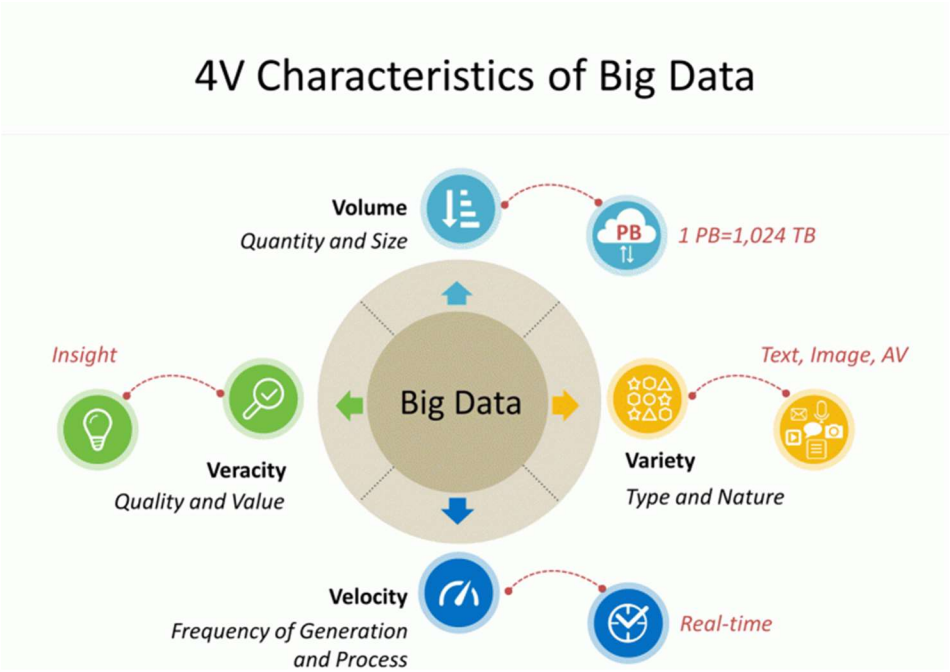


Schéma 1 Big data (zdroj: yourfreetemplates.com)

1.1.6. Nástroje pro práci s Big Data (2)

Při práci s Big data je rozhodující nejen nástroj, který použijeme, ale i technika zpracování.

Pro zpracování dat lze použít rozsáhlé množství jazyků, jejich knihoven či samostatných programů. Mezi nejpopulárnější v souvislosti s Big data v současnosti patří jazyk Python a Apache Spark.

V následující tabulce jsou uvedeny populární nástroje a technologie pro práci s velkým množstvím dat.

Analytics	Data preparation	Data reduction	Data analysis
Tools	R, SAS, Python, Java, C++, SPSS, MATLAB, Minitab	VBA, Excel, MySQL	Apache Hadoop, Apache Spark, GIS, Parallel NetCDF, Javascript, Perl, PHP, Cloud Solutions, AWS, Open source databases
Techniques	Statistics functions, Machine Learning, Factor Analysis, Principal Component Analysis, Natural Language Processing, Neural networks	Linear/Non-linear Regression, Text Analytics	Logical regression, Time-series analysis, Decision tree, clustering, MapReduce, graph analytics

1.2. BIG DATA V GEODÉZII A KARTOGRAFII

1.2.1. Popis

S rychlým rozvojem technologie počítačových sítí jsou Big data již nyní velkou součástí našeho světa, a to ve všech možných odvětvích. Kartografické a geografické informační technologie jsou nyní vyvíjeny tak, aby byly schopny zpracovat opravdu veliké množství dat najednou. Aby uspokojily poptávku po přesných geografických informacích a aby lépe sloužily svému účelu, musí být geografické informace hodnoceny ze všech aspektů. Prohloubení informační hodnoty dat a přizpůsobení se potřebám éry Big dat vyžaduje další průlomy a změny v získávání, zpracování a aplikaci geografických dat.

1.2.2. Rozdělení

Podle vědeckých článků dnes v digitální éře rozlišujeme pět hlavních sekcí v oblasti Big dat v souvislosti s obory geodézie a geomatika:

- Globální geoprostorový průmysl
- GNSS a technologie určování polohy
- GIS/prostorová analýza
- technologie pozorování Země (Earth Observation)
- technologie 3D skenování.

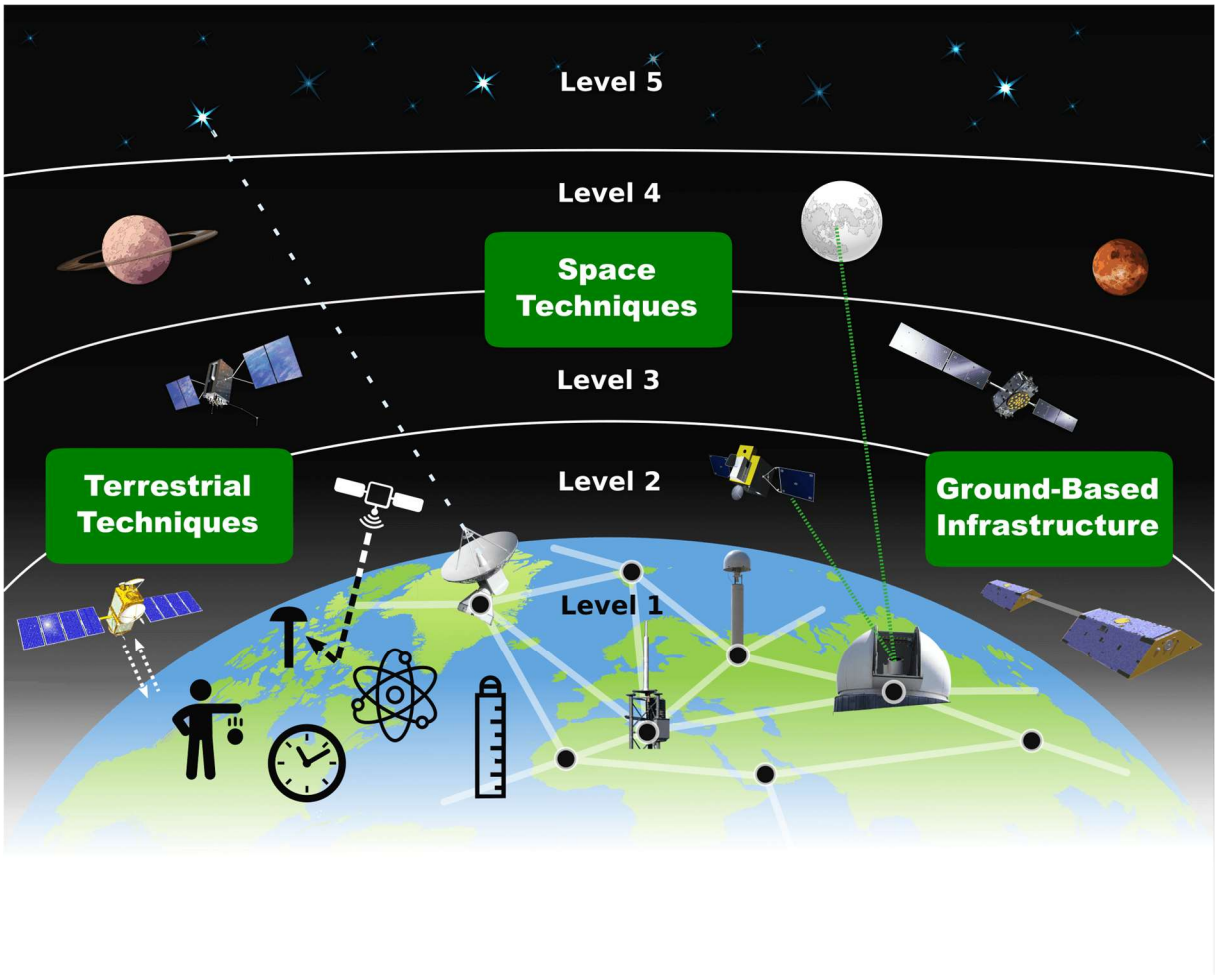


Schéma 2 Zdroje prostorových dat (zdroj: https://ggos.org/obs)

1.2.3. Zdroje

Pokud bychom chtěli sekce zmíněné výše rozebrat podrobněji na zdroje dat/informací, vypadal by výčet asi takto:
3D–AR (rozšířená realita), 3D-VR (virtuální realita), Modelování budov (BIM, GeoBIM, HistoricBIM), Katastrální měření, Kartografie, Monitoring katastrof, Managment(řízení), Technologie pozorování Země (EO), Vědy o Zemi, Inženýrské zeměměřictví, Geovýpočty, Geodézie, Geodetická metrologie (Průmyslová metrologie, 3D-metrologie, velký objem Metrologie, Technická geodézie), Geografie, Geomatika, Geosenzory-Geosensorika (IoT, Edge zařízení), GeoWeb-SensorWeb, GIS a aplikace Časoprostorové analýzy, GIS a Automatické mapování/Facility Management, GNSS a aplikace (PNT, GPS-předpověď počasí, sledování, atd.), Měření s vysokým rozlišením, Hydrografie, Hydrografické měření, Laserové skenování/LIDAR, Lokalizační služby, Fotogrammetrie, Remote sensing, SLAM/SLAMMOT, Smart cities(tzv. chytrá města), Surveying Topography, Bezpilotní prostředky (vzduch, země, voda).... (3)

1.2.4. Prostorová data

Největší část Big dat v oblasti geodézie a geomatiky v dnešní době je tvořena prostorovými (spatial) daty. Prostorová data jsou reprezentována ve formátu těchto základních modelů:

- Rastr (mřížka): satelitní snímky jsou ukázkovým příkladem rastrových dat
- Vektor: skládají se z bodů, čar, mnohoúhelníků a jejich spojení
- Síť: grafy tvořené prostorovými sítěmi jsou dalším důležitým datovým typem používaným k reprezentaci silničních sítí.

Prostorová data jsou zásadní pro jakékoli mapovací aktivity a lze je rozdělit do dvou typů: nezpracovaná (raw) a odvozená. Pro geomorfologické mapování zahrnují nezpracovaná data informace o rozložení výšky, jako jsou vrstevnice a výšky bodů na topografické mapě a rastrovém digitálním výškovém modelu (DEM). V jistém smyslu lze získávání takových údajů o nadmořské výšce nazývat geomorfologické mapování. Kromě toho jsou typickými geomorfologickými mapovými produkty tematické mapy znázorňující prostorové rozložení jednotek tvaru terénu, pro něž se používají jak surová, tak odvozená data. Odvozená data zahrnují derivace DEM, jako je úhel sklonu, zakřivení a poměr stran. Výsledky vizuální interpretace topografických map a leteckých/ družicových snímků jsou rovněž odvozenými daty využitelnými pro aplikované geomorfologické mapování.

Schéma 3 Vrstvy geospatial data (zdroj: U. S. Geological survey)

Další běžná binární klasifikace prostorových dat je analogová versus digitální. Klasická prostorová data jsou v analogovém formátu, jako jsou tištěné mapy a ručně psané ilustrace v terénních poznámkách. Ačkoli analogová data přispěla k rozvoji geomorfologie, kvalitativní a subjektivní povaha těchto dat ztěžuje jejich přímou analýzu pomocí počítačů. Od 80. let 20. století nahradiy digitální mapy do značné míry tradiční analogové zdroje dat a topografické mapy byly nahrazeny DEM, přičemž jejich analýzu usnadnily související technologie, jako jsou rychlé osobní počítače a geografické informační systémy (GIS). (4)

2. DATA ANALYTICS

Stejně jako u téměř každé činnosti je i u zpracování dat důležitý postup. Tato kapitola se krátce zaměřuje na seznámení s analýzou dat jako vědním oborem, jejími kroky a typy.

2.1. Popis

Data analytics (DA) je věda zabývající se zkoumáním nezpracovaných (raw) dat za účelem vyvození závěrů. Jedná se o aplikaci algoritmického, resp. mechanického procesu k získání poznatků. Zaměření DA spočívá v inferenci, což je proces odvozování závěrů, které jsou závislé pouze na tom, co už výzkumník ví. (3)

2.2. Postup zpracování dat

- **Sběr**
  Sběr dat je proces shromažďování informací o objektech či úkazech vztahujících se k vykonávané analýze. Způsob sběru dat by měl zajistit, že shromážděná data jsou přesná, takže učiněná související vyhodnocení jsou platná. Shromažďování dat je základem pro vyhodnocení.
  Data jsou shromažďována z různých zdrojů, od měření v terénu až po informace z organizačních databází. Takto získaná data nemusí být strukturovaná a mohou obsahovat irelevantní informace. Shromážděná data proto musí být podrobena zpracování dat a čištění dat.

- **Organizace dat (Struktura a klasifikace)**
  Shromážděná data musí být zpracována nebo uspořádána pro analýzu. To zahrnuje strukturování dat podle požadavků příslušných analytických nástrojů. Například může být nutné umístit data do řádků a sloupců v tabulce v tabulkovém procesoru nebo statistické aplikaci, či je nutné vytvořit datový model.

- **Čištění**
  Zpracovaná a uspořádaná data mohou být neúplná, mohou obsahovat duplikáty nebo obsahovat chyby. Čištění dat je proces prevence a opravy těchto chyb. Existuje několik typů čištění dat, které závisí na jejich typu. Například při čištění finančních dat mohou být určité součty porovnány se spolehlivými publikovanými čísly nebo definovanými prahovými hodnotami. Podobně lze kvantitativní datové metody použít pro detekci odlehlých hodnot, které by byly následně z analýzy vyloučeny.

- **Zpracování (analýza)**
  Data, která projdou všemi předchozími kroky, tzn. jsou již zpracována, uspořádána a vyčištěna, jsou připravena k analýze. Pro vyvození závěrů jsou k dispozici různé techniky analýzy dat. K prozkoumání dat v grafickém formátu může být také použita vizualizace dat za účelem získání dalšího pohledu na informace v datech obsažené.

  Statistické datové modely, jako je korelace či regresní analýza, lze použít k identifikaci vztahů mezi datovými proměnnými. Tyto modely, které popisují data, jsou užitečné při zjednodušení analýzy a sdělování výsledků. Zpracování může vyžadovat dodatečné čištění či další sběr dat. Všechny předešlé kroky se tedy v podstatě prolínají.

- **Zhodnocení**
  Pro zhodnocení jsou využívány techniky vizualizace dat, jako jsou tabulky a grafy, které pomáhají uživatelům jasně a efektivně sdělit zprávu. Analytické nástroje umožňují zvýraznění požadovaných informací pomocí barevných kódů a formátování v tabulkách a grafech.

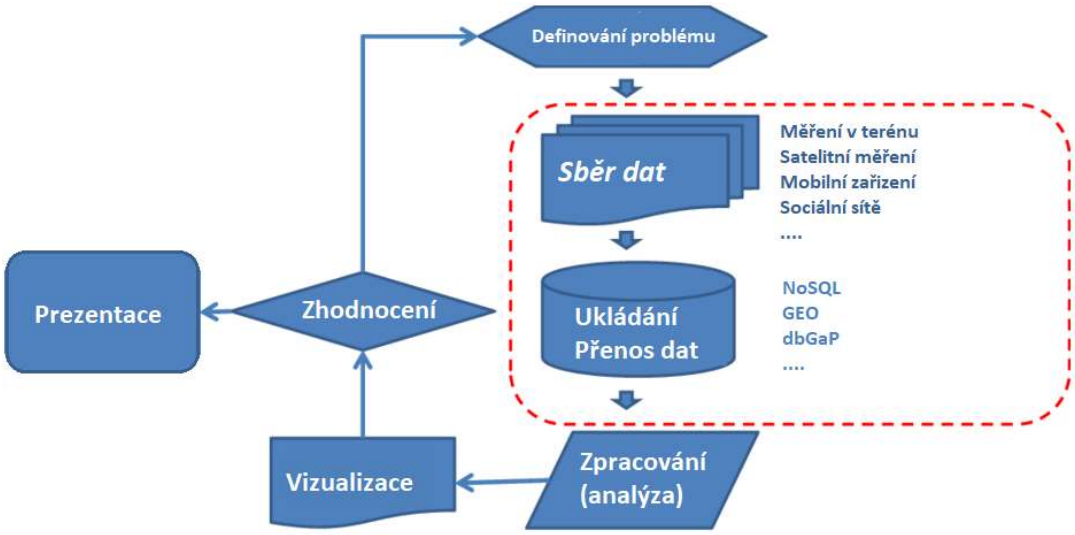


Schéma 4 Analytics workflow

2.3. Typy DA

- **Descriptive**
  Zaměřuje se na přípravu dat pro pokročilou analýzu a zahrnuje různé metody včetně regrese, datového modelování a vizualizace.
  Informuje o minulosti. Pomocí agregace a dolování dat reaguje na otázky: "Co se stalo a proč?" a přispívá ke shrnutí výsledků.
- **Diagnostic**
  Ke studiu používá data z minulosti ke zjištění skutečnosti ze současnosti. Odpovídá na otázky „Proč se toto konkrétně stalo?“, „Co se pokazilo?“
- **Predictive**
  Zahrnuje prediktivní modelování a dolování dat. Zaměřuje se na odvozování predikcí pomocí statistických technik jako je lineární nebo logistická regrese, extrahování datových vzorů pomocí techniky dolování dat.
  Používá poznatky založené na datech z minulosti a současnosti pro předpovídání budoucnosti (událost, trend atd.). Použitím statistického modelování a simulace odpovídá na otázky: "Co se stane?" a přispívá k informovaným rozhodnutím týkajícím se budoucnosti.
- **Prescriptive**
  Zahrnuje především rozhodování a zaměřuje se na efektivitu. Využívá optimalizace a simulace techniky k dosažení efektivity.

Je to nejúčinnější nástroj DA s datovými statistikami používanými k práci s daty za účelem získání informací. Používá modely ke specifikaci optimálního chování, procesů, struktur, systémů a akcí. Pomocí heuristiky a optimalizačních modelů, odpovídá na otázku "Co by se mělo udělat?" a přispívá k složitým a časově omezeným rozhodnutím.

3. NÁSTROJE PRO PRÁCI S DATY

Velká data nelze zpracovávat perem na papír, už jen z důvodu, že jsou dostupná pouze v digitálním světě. Proto je třeba mít na jejich zpracování co nejefektivnější nástroje. Python, jako programovací jazyk je v současnosti nejpoužívanější nástroj pro zpracování Big Dat. Z pohledu analýzy Big Datatoto platí i o Apache Spark (5). Tato kapitola se zabývá popisy těchto platforem pro zpracování velkého množství dat a v případě Apache Spark i jeho předchůdcem. Obsahem této kapitoly je jejich popis, komponenty a interní fungování.

3.1. PYTHON

3.1.1. Základní popis

Python je interpretovaný, objektově orientovaný, vysokoúrovňový programovací jazyk s dynamickou sémantikou. Jeho vestavěné datové struktury na vysoké úrovni v kombinaci s dynamickým psaním a dynamickou vazbou jej činí velmi atraktivním pro rychlý vývoj aplikací a také pro použití jako skriptovacího nebo spojovacího jazyka pro spojení existujících komponent dohromady. Python podporuje moduly a balíčky, což umožňuje opětovné použití kódu. Zahrnuje moduly, výjimky, dynamické psaní, velmi vysoké dynamické datové typy a třídy. Podporuje více programovacích paradigmat nad rámec objektově orientovaného programování, jako je procedurální a funkční programování. Python kombinuje pozoruhodnou sílu s velmi jasnou syntaxí. Má rozhraní pro mnoho systémových volání a knihoven, stejně jako pro různé okenní systémy, a je rozšiřitelný v C nebo C++. Je rovněž použitelný jako rozšiřující jazyk pro aplikace, které potřebují programovatelné rozhraní. Python je přenosný, tzn. běží na mnoha variantách Unix včetně Linuxu a macOS a na Windows. (6)

3.1.2. Knihovny

Jazyk je dodáván s velkou standardní knihovnou, která pokrývá oblasti jako zpracování řetězců (regulární výrazy, Unicode, výpočet rozdílů mezi soubory), internetové protokoly (HTTP, FTP, SMTP, XML-RPC, POP, IMAP, CGI programování), softwarové inženýrství (testování jednotek, protokolování, profilování, parsování kódu Pythonu) a rozhraní operačního systému (systémová volání, souborové systémy, sokety TCP/IP). K dispozici je také široká škála rozšíření třetích stran. V této práci bude Python využíván společně s Apache Spark, který bude popsán v kapitole 3.3. Rozhraní umožňující Pythonu spojení s Apache Spark se nazývá PySpark.

K PySparku se při samotném psaní kódu přistupuje jako ke knihovně. Tzn. musíme ji importovat. V ukázce kódu níže vidíme import SparkSession a všech typů formátů proměnných.

```
from PySpark.sql import SparkSession

from PySpark.sql.types import *
```

PySpark pracuje tzv. líným způsobem(lazy evaluation). Operace jsou tedy zpožděny, dokud není výsledek skutečně požadován. Můžete například napsat úlohu pro načtení datové sady s aplikací řady transformací, ale tyto transformace nebudou provedeny okamžitě. Místo toho se zaznamenají prováděné operace, a jakmile jsou data skutečně potřebná, například při zápisu výsledků, pak se tyto operace aplikují jako jedna celistvá. Tento přístup se používá k tomu, aby se zabránilo stahování celého datového rámce do paměti, a navíc umožňuje efektivnější zpracování napříč clusterem[2].

Jelikož je často vyžadována spolupráce mezi jazyky Python a Java, je v PySparku integrovaná i knihovna Py4J. Ta umožňuje Pythonu dynamicky komunikovat s objekty JVM (Java Virtual Machine). Metody jsou volány tak, jako by se objekty Java nacházely v interpreteru Pythonu a k Java kolekcím (collections) lze přistupovat prostřednictvím standardních metod kolekce Pythonu. Py4J také umožňuje programům Java volat zpět objekty Pythonu. (7)

3.2. APACHE HADOOP

3.2.1. Základní popis

Apache Hadoop je do jisté míry předchůdcem Apache Spark. Je to open source framework, který spravuje zpracování a ukládání dat pro velké datové aplikace ve škálovatelných clusterech počítačových serverů. Nachází se v pomyslném centru ekosystému technologií pracujících s Big data, které se primárně používají pro pokročilé analýzy, včetně prediktivní analýzy, dolování dat a strojového učení. Apache Hadoop je navržený tak, aby byl schopen pracovat s různými formami jak strukturovaných, tak nestrukturovaných dat. Díky tomu může být uživatel při sběru, zpracovaní, analýze a spravování dat flexibilnější, než kdyby se spoléhal pouze na relační databáze či datové sklady.

Schopnost Apache Hadoop zpracovávat a ukládat různé typy dat z něj dělá řešení zvláště vhodné pro prostředí velkých dat. Obvykle zahrnují nejen velké množství dat, ale také směs strukturovaných transakčních dat a polostrukturovaných a nestrukturovaných informací, jako jsou záznamy o kliknutí na internetu, protokoly webových serverů a mobilních aplikací, příspěvky na sociálních sítích, e-maily zákazníků a data senzorů z „Internet of Things“ (IoT).

3.2.2. Apache Hadoop ve smyslu Big dat

Výhoda Apache Hadoop je, že jeho architektura jej předurčuje k fungování převážně na komoditních serverech a možnost se škálovat tak, aby podporoval tisíce hardwarových uzlů. Apache Hadoop Distributed File System (HDFS) je navržen tak, aby poskytoval rychlý přístup k datům napříč uzly v clusteru, a navíc funkce odolné proti chybám, takže aplikace mohou pokračovat v běhu, pokud jednotlivé uzly selžou. Tyto funkce pomohly Apache Hadoop stát se základní platformou pro správu dat při použití v analýze velkých dat poté, co se objevila v polovině 2000.

Protože Apache Hadoop dokáže zpracovat a uložit tak široký sortiment dat, umožňuje organizacím využít rozsáhlá uložiště pro příchozí toky informací, např. Data lake[3]. V data lake Apache Hadoop jsou

[2] Cluster je platforma, na které je naistalován(beží) Apache Spark.
[3] Data lake je centralizované úložiště určené k ukládání, zpracování a zabezpečení velkého množství dat.

nezpracovaná data často uložena tak, jak byla vytvořena, takže vědci a další analytici mají v případě potřeby přístup k úplným sadám dat; data jsou poté filtrována a připravena analytickými nebo IT týmy podle potřeby pro podporu různých aplikací.

Data lake obecně slouží jiným účelům než tradiční datové sklady, které uchovávají vyčištěné sady transakčních dat. V některých případech však společnosti považují svá datová jezera Apache Hadoop za moderní datové sklady. Ať tak či onak, rostoucí role analýzy velkých dat v obchodním rozhodování učinila z efektivního řízení dat a procesů zabezpečení dat prioritu při nasazení datových jezer a systémů Apache Hadoop obecně.

3.2.3. Apache Hadoop součásti

HDFS

Apache Hadoop Distributed File System (HDFS), v překladu distribuovaný souborový systém pro Apache Hadoop, je úložná komponenta, která ukládá data ve formě souborů. Data jsou zapsána jednou na server a následně mnohokrát čtena a použita. Soubory dat jsou rozděleny do samostatných „bloků“, které jsou distribuovány mezi různé datové uzly. Obsahuje architekturu master/ slave. Tato architektura se skládá z jednoho NameNode, který plní roli master, a více DataNode, jež plní roli slave.

NameNode je uzel hlavní a v každém clusteru je pouze jeden. Jeho úkolem je vědět, kde v clusteru se nachází bloky dat.

Datový uzel je uzel podřízený, který ukládá bloky dat. Na každý cluster je jich více než jeden. Jeho úkolem je získávat data, pokud je potřeba. Udržuje neustálý kontakt s uzlem hlavním.

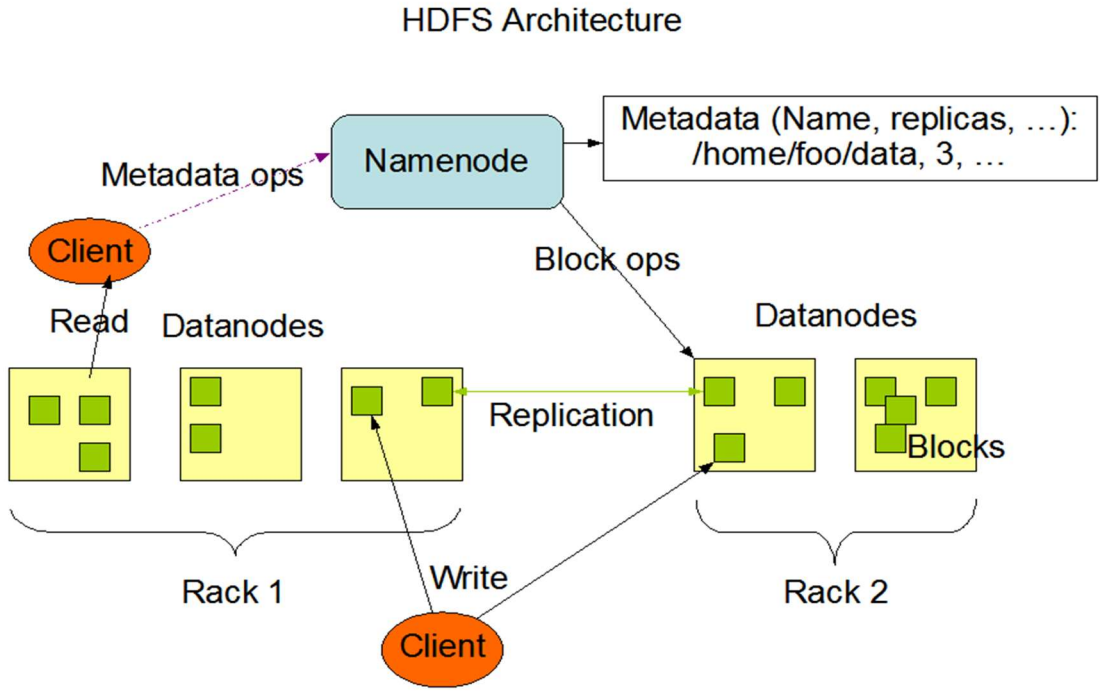


Schéma 5 HDFS (8)

YARN

Apache Hadoop YARN neboli Yet Another Resource Negotiator je technologie pro správu zdrojů a plánování úloh Apache Hadoop. Umožňuje zpracovávat data uložená v HDFS a spouštět je různými enginy pro zpracování dat, jako je batch processing, stream processing, interactive processing a mnoho dalších. YARN, jedna ze základních součástí Apache Hadoop, je tedy odpovědná za přidělování systémových prostředků různým aplikacím běžícím v clusteru Apache Hadoop a plánování úloh, které mají být provedeny na různých uzlech.

Apache Hadoop Common

Balíček Apache Hadoop Common je považován za základ/ jádro rámce, protože poskytuje základní služby a základní procesy. Obsahuje rovněž potřebné soubory Java Archive (JAR) a skripty potřebné ke spuštění Apache Hadoopu. Balíček Apache Hadoop Common poskytuje též zdrojový kód a dokumentaci a také sekci příspěvků, která obsahuje různé projekty z komunity Apache Hadoop.

3.2.4. MapReduce

Pro zpracování velkých dat používá Apache Hadoop algoritmus MapReduce představený společností Google. Ten usnadňuje distribuci úlohy a umožňuje její paralelní spouštění v clusteru. V podstatě rozděluje jeden úkol na více úkolů, které pak zpracovává na různých strojích. Představuje tok dat spíše než proceduru.

Samotná „úloha“ MapReduce se dělí na dvě části. Jsou jimi Map a Reduce. Map filtruje, seskupuje a třídí data. Vstupní data jsou rozdělena. Každá úloha pak pracuje na již rozdělených datech paralelně na různých uzlech a vytváří pár klíč-hodnota. Poté jsou produkty úloh spojeny do ucelené sady výsledků a uloženy na HDFS.

Nabízí se tedy otázka, kdy je vhodné MapReduce použít. MapReduce je vhodný pro iterativní výpočty zahrnující velké množství dat vyžadujících paralelní zpracování.

MapReduce není ideální volbou při:

- potřebě rychlé reakce v řádu jednotek sekund
- real-time zpracování
- zpracování grafů
- implementaci složitých algoritmů, např. některých algoritmů strojového učení, jako je SVM
- iteraci a potřebě data zpracovávat znovu a znovu

Nebo když Map fáze vygeneruje příliš mnoho klíčů a třídění pak může být zdlouhavé, či pokud spojujeme dva velké datové soubory se složitými podmínkami.

Proces MapReduce

Fungování MapReduce si ukážeme na jednoduchém příkladu spočítání celkového výskytu řetězců ve vstupních datech. Níže vidíme data o chybách, které mají označení A, B, C. Tyto vstupy reprezentují rozdílné uzly, na kterých jsou uložena data. Na těch se paralelně provádí úkony až do fáze rozdělení.

Vstup:
Vstup 1: <Chyba A>, <Chyba B>, <Chyba A>, <Chyba C>, <Chyba A>
Vstup 2: <Chyba B>, <Chyba B>, <Chyba A>, <Chyba A>
Vstup 3: <Chyba A>, <Chyba C>, <Chyba A>, <Chyba B>, <Chyba A>
Vstup 4: <Chyba B>, <Chyba C>, <Chyba C>, <Chyba A>

Každý vstup obsahuje záznam (textový řetězec), který se použije jako klíč. V tomto případě jsou to jednotlivé informace o chybách (Chyba A, B, C). Ve fázi map se zpracuje každý záznam tak, aby vytvořil páry klíč-hodnota. Jako hodnotu zde použijeme číslo '1'. Výstup z fáze map vypadá takto:

Map:
Vstup 1: <Chyba A, 1>, <Chyba B, 1>, <Chyba A, 1>, <Chyba C, 1>, <Chyba A, 1>
Vstup 2: <Chyba B, 1>, <Chyba B, 1>, <Chyba A, 1>, <Chyba A, 1>
Vstup 3: <Chyba A, 1>, <Chyba C, 1>, <Chyba A, 1>, <Chyba B, 1>, <Chyba A, 1>
Vstup 4: <Chyba B, 1>, <Chyba C, 1>, <Chyba C, 1>, <Chyba A, 1>

Při fázi sloučení se sečtou všechny hodnoty, v tomto případě výskyty, jednotlivých klíčů v každém vstupu zvlášť.

Sloučení:
Combiner 1: <Chyba A, 3>, <Chyba B, 1>, <Chyba C, 1>
Combiner 2: <Chyba A, 2> <Chyba B, 2>
Combiner 3: <Chyba A, 3> <Chyba B, 1> <Chyba C, 1>
Combiner 4: <Chyba A, 1> <Chyba B, 1> <Chyba C, 2>

Poté rozdělovač přiřadí data ze slučovačů do reduktorů. Tzn. data se sloučí podle klíčů a k nim se přiřadí počet výskytů v každém původním vstupu.

Rozdělení:
Reducer 1: <Chyba A> {3,2,3,1}
Reducer 2: <Chyba B> {1,2,1,1}
Reducer 3: <Chyba C> {1,1,2}

Pokud by nebyly zapojeny žádné slučovače, vstup do reduktorů by byl následující:

Reducer 1: <Chyba A> {1,1,1,1,1,1,1,1,1}
Reducer 2: <Chyba B> {1,1,1,1,1}
Reducer 3: <Chyba C> {1,1,1,1}

Thákurova 7, 166 29 Praha 6

V tomto případě na přítomnosti slučovače nezáleží. Pokud by se ale jednalo o terabajty dat, přítomnost slučovače je v podstatě nezbytná.

Nyní každý reduktor pouze vypočítá celkový počet výskytů. Tzn. pro každý unikátní klíč se spočte celkový počet výskytů.

Reduce:
Reducer 1: <Chyba A, 9>
Reducer 2: <Chyba B, 5>
Reducer 3: <Chyba C, 4>

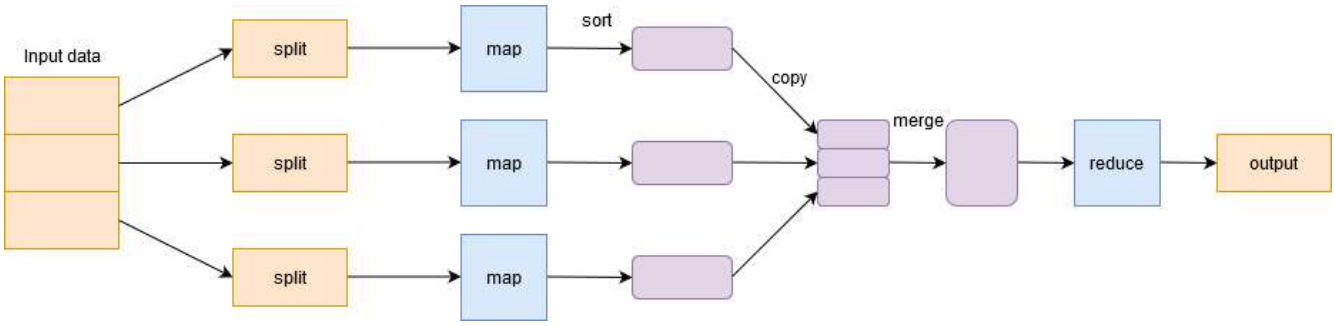


Schéma 6 MapReduce (9)

3.2.5. Historie a vývoj

Základní komponenty první verze Apache Hadoop byly MapReduce, HDFS a Apache Hadoop Common, což je sada sdílených pomocných programů a knihoven. MapReduce zpočátku fungoval jako zpracovatelský stroj Apache Hadoop a správce clusterových prostředků, který s ním přímo svázal HDFS a omezoval uživatele na spouštění dávkových aplikací MapReduce.

To se změnilo ve verzi Apache Hadoop 2.0, která se stala obecně dostupnou v říjnu 2013, kdy byla vydána verze 2.2.0. V té představil Apache Hadoop YARN, novou technologii pro správu prostředků clusteru a plánování úloh, která převzala tyto funkce od MapReduce. YARN – zkratka pro Yet Another Resource Negotiator, ale obvykle označovaná pouze zkratkou – ukončila závislost na MapReduce a otevřela Apache Hadoop dalším zpracovatelským enginům a různým aplikacím kromě dávkových úloh. Apache Hadoop nyní může například spouštět aplikace na enginech Apache Spark, Apache Flink, Apache Kafka a Apache Storm.

V clusterech Apache Hadoop sedí YARN mezi HDFS a procesory nasazenými uživateli. Správce prostředků používá kombinaci kontejnerů, aplikačních koordinátorů a agentů monitorování na úrovni uzlů k dynamickému přidělování clusterových prostředků aplikacím a dohlíží na provádění úloh

zpracování v decentralizovaném procesu. YARN podporuje více přístupů k plánování úloh, včetně fronty „first-in-first-out“ a několika metod, které plánují úlohy na základě přidělených zdrojů clusteru.

Řada vydání Apache Hadoop 2.0 také přidala funkce pro vysokou dostupnost a federaci pro HDFS, podporu pro spouštění clusterů Apache Hadoop na serverech Microsoft Windows a další funkce určené k rozšíření všestrannosti rámce distribuovaného zpracování pro správu a analýzu velkých dat.

Apache Hadoop 3.0.0 byl další hlavní verzí Apache Hadoop. Vydaný Apache v prosinci 2017 a přidal funkci YARN Federation navrženou tak, aby umožnila YARN podporovat desítky tisíc uzlů nebo více v jednom clusteru, oproti předchozímu limitu 10 000 uzlů. Nová verze také obsahovala podporu pro GPU a kódování pro vymazání, což je alternativa k replikaci dat, která vyžaduje výrazně méně úložného prostoru.

Následné aktualizace 3.1.x a 3.2.x umožnily uživatelům Apache Hadoop spouštět kontejnery YARN v kontejnerech Docker a představily rámec služeb YARN, který funguje jako platforma pro orchestraci kontejnerů. S těmito verzemi byly také přidány dvě nové součásti Apache Hadoop: stroj pro strojové učení s názvem Apache Hadoop Submarine a úložiště objektů Apache Hadoop Ozone, které je postaveno na vrstvě blokového úložiště Apache Hadoop Distributed Data Store a je navrženo pro použití v místních systémech.

3.2.6. Implementace a využití

YARN výrazně rozšířil aplikace, které mohou clustery Apache Hadoop zpracovávat, aby zahrnovaly interaktivní dotazování, zpracování streamů a analýzy v reálném čase. Například výrobci, energetické společnosti, ropné a plynárenské společnosti a další podniky používají data v reálném čase, která se do systémů Apache Hadoop přenášejí ze zařízení IoT v aplikacích prediktivní údržby, aby se pokusili odhalit poruchy zařízení dříve, než k nim dojde. Detekce podvodů, personalizace webových stránek a hodnocení zákaznické zkušenosti jsou další případy použití v reálném čase.

Další obvyklé příklady užití Apache Hadoop:

- Zákaznická analytika. Příklady zahrnují snahy předvídat odchod zákazníků, analyzovat data o kliknutích pro lepší cílení online reklam na uživatele webu a sledovat sentiment zákazníků na základě komentářů o společnosti na sociálních sítích.
- Řízení rizik. Společnosti poskytující finanční služby využívají clustery Apache Hadoop k vývoji přesnějších modelů analýzy rizik pro interní použití i pro jejich zákazníky. Vytvářejí také investiční modely a vyvíjejí obchodní algoritmy v systémech velkých dat založených na Apache Hadoop.
- Operační zpravodajství. Apache Hadoop může například pomoci telekomunikačním společnostem lépe porozumět výkonu přepínání a využití sítě a frekvence pro plánování a správu kapacity. Analýzou spotřeby mobilních služeb a dostupné šířky pásma

v geografických oblastech, mohou telekomunikační společnosti také určit nejlepší místa pro umístění nových mobilních věží a rychleji reagovat na problémy se sítí.

- Řízení dodavatelského řetězce. Výrobci, maloobchodníci a přepravci používají systémy Apache Hadoop ke sledování pohybu zboží a vozidel, aby mohli určit náklady na různé možnosti přepravy. Kromě toho mohou analyzovat velké množství historických dat o poloze s časovou značkou, aby zmapovali potenciální zpoždění a optimalizovali trasy doručení.

Technologie byla nasazena i pro mnoho dalších použití. Pojišťovny například používají Apache Hadoop pro aplikace, jako je analýza cenové politiky a správa programů slev pro bezpečné řidiče. Zdravotnické organizace zase hledají způsoby, jak s pomocí Apache Hadoop zlepšit léčbu pacientů.

3.2.7. Ukázky

Ukázky v teoretické části jsou převzaty z výukového kurzu na udemy.com s názvem Apache Spark with Scala – Hands On with Big data! (10). Ukázky v teoretické části jsou tedy psány v jazyce Scala, ve kterém je Apache Spark napsán. Více o Apache Spark v následující kapitole.

Apache Hadoop MapReduce pomocí Apache Spark – Word Count aplikace

Tato aplikace počítá výskyt každého unikátního slova(řetězce) v zadaném textovém souboru.

```
/** Import potřebných knihoven */
import org.apache.spark._
import org.apache.log4j._

/** Výpočet úkazů jednotlivých slov v dokumentu. */
object WordCount {

 /** Main – zde se provádí příkaz */
 def main(args: Array[String]) {


// Vytvoření SparkContextu
// local[*] – využití všech dostupných jader procesoru. Při definování přesného počtu se * nahradí číselnou hodnotou
// "WordCount" – název aplikace
   val sc = new SparkContext("local[*]", "WordCount")

   // Čtení dokumentu řádek po řádku a transformace do RDD
   val input = sc.textFile("source.txt")

   // Nastavení oddělovače
  // Na proměnné input se provede transformace pomocí flatMap a za separační proměnnou se použije mezera
```

```
  val words = input.flatMap(x => x.split(" "))

  // Samotný výpočet úkazů jednotlivých slov
  //Na proměnné RDD se použije funkce countByValue
  val wordCounts = words.countByValue()

  //Tisk výsledků do konzole
  wordCounts.foreach(println)
 }
}
```

3.3. SPARK

3.3.1. Základní popis

Apache Spark je open-source systém zpracování používaný pro velké objemy dat. Využívá ukládání do mezipaměti a optimalizované provádění dotazů pro rychlé analytické dotazy na data jakékoli velikosti. Poskytuje vývojová rozhraní API v jazycích Java, Scala, Python a R a podporuje opětovné použití kódu – dávkové zpracování, interaktivní dotazy, analýzy v reálném čase, strojové učení a zpracování grafů. (5)

Apache Spark je nástupcem již zmíněného Apache Hadoop. I přes to však Apache Spark přebírá velké množství knihoven původně vytvořených pro Apache Hadoop. Největšími výhodami Apache Spark jsou rychlost, API pro ostatní již zmíněné jazyky a větší množství funkcí.

Obecně je používání komoditního hardwaru pro analýzu velkých dat v cloudu široce rozšířeno. Proto si Apache Hadoop MapReduce a jeho nástupci (Apache Spark, …) získali popularitu díky vysoké škálovatelnosti, odolnosti proti chybám a nízkým nákladům na infrastrukturu.

V současné době je Apache Hadoop postaven na předpokladu, že dotaz je rozdělen a zpracován jako několik samostatných úloh a Apache Hadoop čte data z disku a ukládá mezivýsledky rovněž na disk. Hlavním účelem tohoto postupu je zaručit odolnost vůči chybám. Pokud je stejná datová sada dotazována opakovaně, je tak prováděno přímo z disku. V případě Apache Spark lze mezivýsledky uložit do paměti. V případě velkých datových sad se může dosáhnout až několika terabajtů nebo dokonce petabajtů a vždy bude problém, když dojde paměť. Ukládání mezivýsledků do paměti se považuje za jakousi pomoc. Pokud se paměť vyčerpá pro všechny oddíly datové sady v lokalitě uzlů, pak se RDD[4] uloží na disk. Další možností je přepočítat datovou sadu v momentě, kdy je znovu potřeba. Toto může být problém pro správu paměti v případě, kdy mnoho uživatelů využívá sdílenou paměť clusteru.

Jak již bylo zmíněno v kapitole 3.1.2, pro využití Apache Spark společně s Pythonem je klíčová jedna extenze, a tou je PySpark. PySpark je API (Application Programming Interface). Umožňuje psát

[4] RDD je základní datová struktura Apache Spark. Popis RDD v kapitole 3.3.8.

aplikace Spark pomocí Pythonu a poskytuje prostředí pro interaktivní analýzu dat. Podporuje většinu funkcí Spark jako SparkSQL, Streaming, MLlib atd. Více o uvedených součástech v další kapitole.

3.3.2. Jádro

Jádro je část Apache Spark, spojující všechny ostatní Apache Spark systémy dohromady. Spolu se správcem clusteru spravuje a distribuuje úlohy mezi ostatní clustery v síti. Zároveň poskytuje odolnost proti poruchám, základní I/O operace, které umožňují kooperaci s datovými uložišti a spravování paměti a RDD.

Součásti

1. **Apache Spark Core**
   Spark Core je základním obecným prováděcím enginem pro platformu Spark, na kterém jsou postaveny všechny ostatní funkce. Poskytuje výpočty v paměti a referenční datové sady v externích úložných systémech.

   Spark Core je tedy výkonný engine pro platformu Spark, který je vyžadován a používán jinými součástmi, které jsou postaveny na Spark Core. Spark Core poskytuje výpočetní a referenční datové sady uložené v externích úložných systémech vestavěné paměti. Hlavní odpovědností Sparku je provádět všechny základní I/O funkce, plánování, monitorování atd. Dalšími důležitými funkcemi Spark Core jsou také obnova chyb a efektivní správa paměti.

   Spark Core používá velmi speciální datovou strukturu zvanou RDD. Sdílení dat v systémech distribuovaného zpracování, jako je MapReduce, vyžaduje, aby byla data uložena v mezikrocích a poté načtena z trvalého úložiště, jako je HDFS nebo S3, což jej velmi zpomaluje kvůli serializaci a deserializaci I/O kroků. RDD tento problém řeší tak, že tyto datové struktury jsou v paměti a jsou tedy odolné proti chybám a lze je sdílet napříč různými úkoly v rámci stejného procesu. RDD mohou být jakkoliv neměnné či rozdělené kolekce a mohou obsahovat jakýkoli typ objektů: Python, Scala, Java nebo jakékoliv uživatelem definované objekty třídy. RDD lze vytvořit buď transformací existujícího RDD nebo načtením z externích zdrojů, jako je HDFS nebo HBase.

2. **Spark SQL**
   Spark SQL je modul Apache Spark pro práci se strukturovanými daty. Rozhraní nabízená Spark SQL poskytují Sparku více informací o struktuře dat i prováděných výpočtů.

   Spark SQL je postaven na Sharku, který byl prvním interaktivním SQL na systému Apache Hadoop. Shark byl postaven na základně Hive a dosáhl zlepšení výkonu pomocí fyzického spouštěcího enginu Hive. Ale kvůli omezením Hive nebyl Shark schopen dosáhnout takového výkonu, který byl požadován. Projekt Shark byl tedy zastaven a Spark SQL byl postaven se znalostí z projektu Shark na Spark Core Enginu.

Spark SQL se jmenuje podle toho, že pracuje s daty podobným způsobem jako SQL. Ve skutečnosti existuje zmínka, že cílem Spark SQL je splnit standardy SQL 92. Podstatou však je, že umožňuje vývojářům psát deklarativní kód, který umožňuje enginu využívat co nejvíce dat a uložené struktury (RDD) k optimalizaci výsledného distribuovaného dotazu. Cílem je umožnit uživateli, aby se nemusel tolik starat o distribuci dat a ostatní problémy a mohl se zaměřit na aplikaci a využití ve svém pracovním odvětví. Uživatelé mohou provádět funkce extrakce, transformace a načítání dat z různých zdrojů v různých formátech, jako je JSON, Parquet nebo Hive, a poté provádět dotazy pomocí Spark SQL (např. *DF.select("sloupec_1", " sloupec _2", " sloupec _3"))*

DataFrame tvoří hlavní abstrakci pro Spark SQL. Distribuovaná kolekce dat uspořádaných do pojmenovaných sloupců je ve Sparku známá jako DataFrame. V dřívějších verzích Spark SQL byly DataFrames označovány jako SchemaRDD. DataFrame API ve Sparku se integruje s procedurálním kódem Spark, aby byla zajištěna integrace mezi procedurálním a relačním zpracováním. Více v kapitole 3.3.8. DataFrame API vyhodnocuje operace tzv. líným způsobem, aby poskytlo podporu pro relační optimalizace a byl maximálně využit celkový pracovní tok zpracování dat.

Catalyst, rozšiřitelný optimalizátor, je jádrem fungování Spark SQL. Optimalizační rámec zabudovaný do jazyku Scala, který pomáhá vývojářům zlepšit produktivitu a výkon dotazů. Pomocí Catalystu mohou vývojáři Spark stručně specifikovat složité relační optimalizace a transformace dotazů v několika řádcích kódu tím, že co nejlépe využívají výkonné programovací konstrukce Scaly, jako je porovnávání vzorů a runtime metaprogramování. Catalyst usnadňuje proces přidávání optimalizačních pravidel, zdrojů dat a datových typů pro domény strojového učení.

Ukázka Spark SQL

Tato aplikace vybírá(vyhledává) pomocí SQL příkazu osoby, kterým je 13-19 let ze zadaného seznamu. Seznam obsahuje id, jméno, věk a počet přátel.

```
/** Import potřebných knihoven */

import org.apache.spark.sql._
import org.apache.log4j._

/** Ukázka operací SparkSQL*/
object SparkSQL {


/** Vytvoření vlastní classy, case classy se používají pro meněnná data */
  case class Person(ID:Int, name:String, age:Int, numFriends:Int)

  def Vstup(line:String): Person = {
```

```
    val fields = line.split(',')

    val person:Person = Person(fields(0).toInt, fields(1), fields(2).toInt, fields(3).toInt)
    person }
  /** Main – zde se vykonává příkaz */
   def main(args: Array[String]) {

    // Použití SparkSession, kterou představil Spark 2.0
    val spark = SparkSession
     .builder
     .appName("SparkSQL")
     .master("local[*]")
     .getOrCreate()


     // Čtení csv souboru s dazy
    val lines = spark.sparkContext.textFile("data.csv")
    val people = lines.map(Vstup)

    // Inference schematu, registrace DataSetu jako tabulky.
    import spark.implicits._
    val schemaPeople = people.toDS

    schemaPeople.printSchema()

    schemaPeople.createOrReplaceTempView("people")

    // SQL dotazy mohou být použity na DataFrame, pokud byl registrován jako tabulka
    val teenagers = spark.sql("SELECT * FROM people WHERE age >= 13 AND age <= 19")

    val results = teenagers.collect()

    results.foreach(println)

    spark.stop()
   }
  }
```

3. **Spark Streaming**
   Tato komponenta umožňuje Sparku zpracovávat streamovaná data v reálném čase. Data lze přijímat z mnoha zdrojů, jako je Kafka, Flume a HDFS (Apache Hadoop Distributed File System). Poté mohou být data zpracována pomocí složitých algoritmů a přenesena do souborových systémů nebo databází.

Jedná se o velmi oblíbenou knihovnu Spark, protože využívá velký výkon Sparku pro zpracování velkých dat a zvyšuje rychlost přenosu. Spark Streaming má schopnost streamovat gigabajty za sekundu. Spark Streaming se používá pro analýzu nepřetržitého toku dat. Běžným příkladem je například zpracování LOG z webu nebo serveru.

Spark streaming není z technického pohledu streamování. Ve skutečnosti rozděluje data na jednotlivé části, které zpracovává společně jako RDD. Nezpracovává tedy data jako bajty najednou, jak přicházejí, ale zpracovává je každou sekundu či jiný pevný časový interval. Pokud to shrneme tak streamování Spark není v reálném čase, ale téměř v reálném čase nebo mikrodávkování, ovšem pro použití většinou aplikací je více než dostačující.

Streamování Spark lze nakonfigurovat tak, aby komunikovalo s různými zdroji dat. Můžeme tedy jen poslouchat port, na který přichází hromada dat, nebo se můžeme připojit ke zdrojům dat, jako je Amazon Kinesis, Kafka, Flume atd. Pro připojení Sparku k těmto zdrojům jsou k dispozici konektory. Spark streaming je vyhledávanou možností nejen pro svou rychlost, ale i pro svou spolehlivost. Má koncept zvaný „kontrolní bod“, který periodicky ukládá stav na disk a v závislosti na tom, jaké zdroje dat nebo jaký přijímač používáme, dokáže obnovit data z bodu, kdy došlo k selhání. Jedná se o velmi robustní mechanismus pro zvládnutí všech druhů selhání, jako je selhání disku nebo selhání uzlu atd. Spark Streaming má garanci přesně jedné zprávy (každá zpráva je doručena přesně jednou) a pomáhá obnovit ztracenou práci, aniž by bylo nutné psát další kód nebo přidávat další konfigurace.

Stejně jako Spark SQL má koncept Dataframe/Dataset postavený na RDD, má Spark streaming něco, co se nazývá Dstream. Jedná se o kolekci RDD, která ztělesňuje celý datový proud. Na Dstream lze aplikovat většinu vestavěných funkcí na RDD, jako je Flatmap, map atd. Dstream lze také rozdělit na jednotlivé RDD a zpracovávat je po částech.

Spark Streaming

Tato aplikace se přes vývojářský účet připojí na platformu Twitter a po daný čas počítá hodnoty výskytu nejvíce používaných hashtagů.

```
/** Import potřebných knihoven */

import org.apache.spark.streaming._
import org.apache.spark.streaming.twitter._

/** Připojení se na Twitter a sledování nejpopulárnějších hashtagů za daný čas
 */
object PopularHashtags {

 /** Konfigurace služeb Twitteru pomocí twitter.txt souboru */
```

```
  def setupTwitter(): Unit = {
   import scala.io.Source

   val lines = Source.fromFile("data/twitter.txt")
   for (line <- lines.getLines) {
    val fields = line.split(" ")
    if (fields.length == 2) {
     System.setProperty("twitter4j.oauth." + fields(0), fields(1))
    }
   }
   lines.close()
  }

/** Main - zde se vykonává příkaz */
  def main(args: Array[String]) {

   // Konfigurace pověření Twitteru pomocí twitter.txt souboru
   setupTwitter()

   // Nastavení Spark streaming contextu se jménem "PopularHashtags", který se spustí
lokálně a využije všech jader procesoru
   // Dávkování dat po jedné sekundě, tzn. data budou přicházet v packetech s intervalem 1s
   val ssc = new StreamingContext("local[*]", "PopularHashtags", Seconds(1))

   // Vyvarování se spamu v konzoli (Standardně by to bylo provedeno mimo funkci main, ale
u streamingu je to vyžadováno až po vytvoření StreamingContextu)
   setupLogging()

   // Vytvoření DStreamu z Twitteru pomocí námi vytvořeného StreamingContextu
   val tweets = TwitterUtils.createStream(ssc, None)

   // Extrahování každého status updatu do DStreams pomocí funkce map()
   val statuses = tweets.map(status => status.getText)

   // Rozložení statusů na mezery pomocí separátoru
   val tweetwords = statuses.flatMap(tweetText => tweetText.split(" "))

   // Selekce hashtagů
   val hashtags = tweetwords.filter(word => word.startsWith("#"))

   // Transformace pomocí funkce map() na formát (hashtag, 1). Pomocí tohoto poté pouze
spočteme počet jedniček u každého hashtagu
```

```
    val hashtagKeyValues = hashtags.map(hashtag => (hashtag, 1))

    // Počítání výskytu hashtagů v rozmezí pěti minut s intervalem jedné sekundy
    val hashtagCounts = hashtagKeyValues.reduceByKeyAndWindow( (x,y) => x + y, (x,y) => x -
y, Seconds(300), Seconds(1))
    //  Předchozí příkaz lze také napsat takto:
    //val hashtagCounts = hashtagKeyValues.reduceByKeyAndWindow( _ + _, _ -_,
Seconds(300), Seconds(1))

    // Seřazení výsledků podle počtu výskytů
    val sortedResults = hashtagCounts.transform(rdd => rdd.sortBy(x => x._2, ascending =
false))

    // Tisk výsledků do konzole
    sortedResults.print

    // Nastavení checkpoint složky a spuštění
    ssc.checkpoint("C:/checkpoint/")
    ssc.start()
    ssc.awaitTermination()
  }
}
```

4. **MLlib (Machine Learning Library)**

MLlib je jednou z mnoha knihoven Apache Spark. Tato knihovna obsahuje širokou škálu algoritmů strojového učení – klasifikaci, regresi, shlukování a kolaborativní filtrování. Zahrnuje také další nástroje pro konstrukci, vyhodnocování a ladění ML Pipelines. Všechny tyto funkce pomáhají Sparku škálovat v celém clusteru.

Dnes se mnoho společností zaměřuje na vytváření datových produktů a služeb zaměřených na zákazníka, které potřebují strojové učení k vytváření prediktivních přehledů, doporučení a personalizovaných výsledků. Datoví vědci mohou tyto problémy vyřešit pomocí populárních jazyků jako Python a R, ale tráví spoustu času budováním a podporou infrastruktury pro tyto jazyky. Spark má vestavěnou podporu pro strojové učení a datovou vědu v masivním měřítku pomocí clusterů. MLLib je zkratka pro Machine Learning Library.

MLlib je nízkoúrovňová knihovna strojového učení. Lze ji volat z programovacích jazyků Java, Scala a Python. Je snadno použitelná, škálovatelná a lze ji snadno integrovat s dalšími nástroji a frameworky. MLlib usnadňuje nasazení a vývoj škálovatelných kanálů strojového učení.

Strojové učení pomáhá naprogramovat systémy tak, aby se automaticky učily a zlepšovaly se zkušenostmi. Aplikace technik strojového učení na Big data je jádro datové analýzy. Statistiky získané použitím různých algoritmů ML mají zásadní roli ve zlepšování podnikání. Některé z příkladů, kde je vidět velký přínos a význam algoritmů ML v poslední době jsou: personalizovaná léčba pomocí analýzy lékařských záznamů pacientů, navigační systémy, které fungují na základě předchozích dat ze senzorů, doporučovací systémy, které řídí prodej v odvětví elektronického obchodování, klasifikátory zabraňující spamu v e-mailu, systémy doporučování, návrhy přátel na sociálních sítích atd.

5. **GraphX**

Apache Spark také přichází s knihovnou pro manipulaci s databázemi grafů a provádění výpočtů nazvanou GraphX. GraphX sjednocuje proces ETL (Extract, Transform, and Load), průzkumnou analýzu a iterativní grafové výpočty v rámci jednoho systému.

GraphX je užitečný při poskytování celkových informací o síti grafů, protože dokáže zjistit, kolik trojúhelníků se v grafu objeví, a aplikovat na něj algoritmus PageRank. Může měřit věci jako „propojenost“, rozložení stupňů, průměrnou délku cesty a další měření na vysoké úrovni grafu. Dokáže také spojovat grafy dohromady a rychle je transformovat. Podporuje také Pregel API pro procházení grafu. Spark GraphX poskytuje Resilient Distributed Graph (RDG – abstrakce Spark RDD). Rozhraní API RDG používají datoví vědci k provádění několika operací s grafy prostřednictvím různých výpočetních primitiv. Podobně jako základní operace u RDD, např. map či filtr, se i grafy vlastností skládají ze základních operátorů. Tyto operátory berou UDF (uživatelem definované funkce) a vytvářejí nové grafy. Navíc jsou vyráběny s transformovanými vlastnostmi a strukturou.

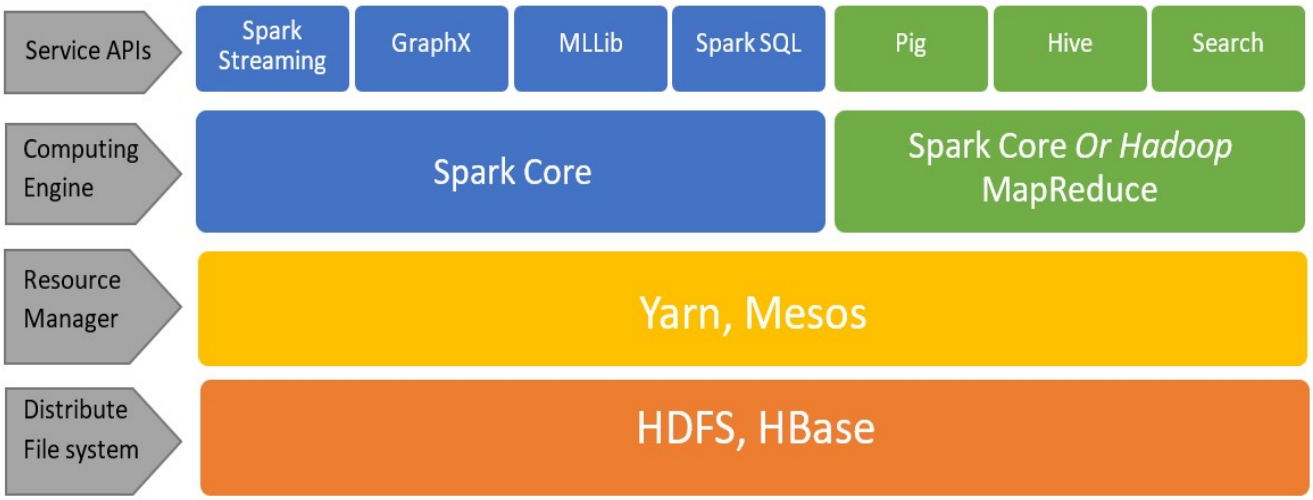


Schéma 7 Apache Spark Ecosystem (11)

3.3.3. Správce clusteru

Jejich prací je přidělení zdrojů pro provedení dotazů/ úloh. Driver může dávat pokyny pouze uzlům, které byly správcem clusteru vybrány. Apache Spark v současné době podporuje čtyři typy správců clusteru: Standalone, Apache Mesos, Apache Hadoop Yarn a Kubernetes.

Standalone
Standalone je jednoduchý správce clusteru integrovaný ve Sparku. Usnadňuje nastavení clusteru a je většinou nejjednodušší cestou, jak spustit Spark aplikaci na clusteru. V tomto módu přiděluje Spark prostředky na základě jádra. Ve výchozím nastavení aplikace využije všech dostupných zdrojů (jádra procesoru a operační paměť) v clusteru.

Apache Mesos
Mesos řeší pracovní zátěž v distribuovaném prostředí dynamickým sdílením a izolací zdrojů. Je vhodný pro nasazení a správu aplikací v prostředích velkých clusterů. Apache Mesos spojuje existující zdroje uzlů v clusteru do jednoho virtuálního zdroje. V jistém smyslu je Apache Mesos opakem virtualizace. A to proto, že při virtualizaci se jeden fyzický zdroj rozdělí na mnoho virtuálních zdrojů. Zatímco v Mesos je mnoho fyzických zdrojů sloučeno do jediného virtuálního zdroje.

Apache Hadoop Yarn
YARN neboli Yet Another Resource Negotiator spravuje prostředky v clusteru a spravuje aplikace přes Apache Hadoop. Umožňuje zpracovávat data uložená v HDFS a spouštět je různými enginy pro zpracování dat, jako je batch processing, stream processing, interactive processing a mnoho dalších.

- Resource manager YARN spravuje zdroje mezi všemi aplikacemi v systému. Správce prostředků má plánovač a Správce aplikací. Plánovač přiděluje prostředky různým spuštěným aplikacím. Je to čistý plánovač, provádí monitorování nebo sledování stavu aplikace. Správce aplikací spravuje aplikace napříč všemi uzly.
- Node manager YARN obsahuje aplikační master a container. Container je místo, kde se odehrává práce. Každá úloha MapReduce běží v jednom containeru. Poskytuje informace o daném uzlu.

Kubernetes
Kubernets je open-source systém pro automatizaci nasazení, škálování a správu kontejnerových aplikací. Kubernets, byť zmíněný v dokumentaci, není v současné době obecně používaný.

3.3.4. Architektura clusteru

Architektura Apache Spark je založená na principu master/ slave (worker).

Driver
Driver je program, ve kterém je vytvořen SparkContext a obsahuje metodu Main().

Vytváří a zadává úlohy pro zpracování, které pak zpracovává slave (worker) uzel. Může běžet na jakémkoliv stroji, který je adresovaný slave uzly v síti a je tedy nezávislý jak na slave, tak na master uzlech. Rovněž je nezávislý na fyzické vzdálenosti od clusteru. Je ale doporučeno v reálném použití mít driver ve fyzické blízkosti sítě, na které je cluster, aby se minimalizovalo zpoždění způsobené přenosem.

Client – V režimu klient běží driver na stroji, na kterém je spuštěn. Z tohoto stroje udržuje SparkContext, dokud provádění dotazu neskončí. Dotaz je tedy závislý na stroji, na kterém je spuštěn, a v případě problému v místním stroji se úloha ukončí. Je vyžadováno, aby byl klient po celou dobu v kontaktu s clusterem, tudíž klient musí být online/ připojen na cluster, dokud se dotaz nevykoná. Výsledky dotazů lze zobrazit bez nutnosti je ukládat na disk s možností zobrazení informací o probíhajících i hotových dotazech v clusteru přímo ve webovém rozhraní. Užívá se pro potřeby ladění a není vhodný pro produkční/ komerční použití.

Cluster – V režimu cluster běží driver přímo v clusteru na jednom ze slave uzlů. Po odeslání dotazu tedy může klient přerušit spojení a „odejít“. O nasazení na konkrétní slave uzel rozhoduje master uzel podle zdrojů a specifikací daných uzlů. Používá se při spouštění dotazů z velké fyzické vzdálenosti, kvůli minimalizaci zpoždění způsobeného přenosem. Nebo v případě, pokud úloha poběží dlouhou dobu a nechceme čekat na výsledek, odešleme ji v režimu clusteru, aby klient nemusel být online.

Správce clusteru – Primární úlohou správce clusteru je rozdělování práce mezi jednotlivé slave uzly.

Jejich prací je tedy alokace zdrojů pro provedení dotazů/ úloh. Driver může dávat pokyny pouze uzlům, které byly správcem clusteru vybrány. Apache Spark v současné době podporuje čtyři typy správců clusteru: Standalone, Apache Mesos, Apache Hadoop Yarn a Kubernetes.

Slave (worker) – Výpočetní uzel v clusteru, který provádí jednotlivé úlohy/ dotazy.

Executor – Exekutoři jsou procesy slave (worker) uzlů, které mají na starosti spouštění jednotlivých úloh v dané úloze. Spouštějí se na začátku aplikace a obvykle běží po celou dobu životnosti. Po spuštění úlohy odešlou výsledky zpět na driver. Poskytují také úložiště v paměti pro RDD, které jsou ukládány do mezipaměti uživatelských programů prostřednictvím Správce bloků.

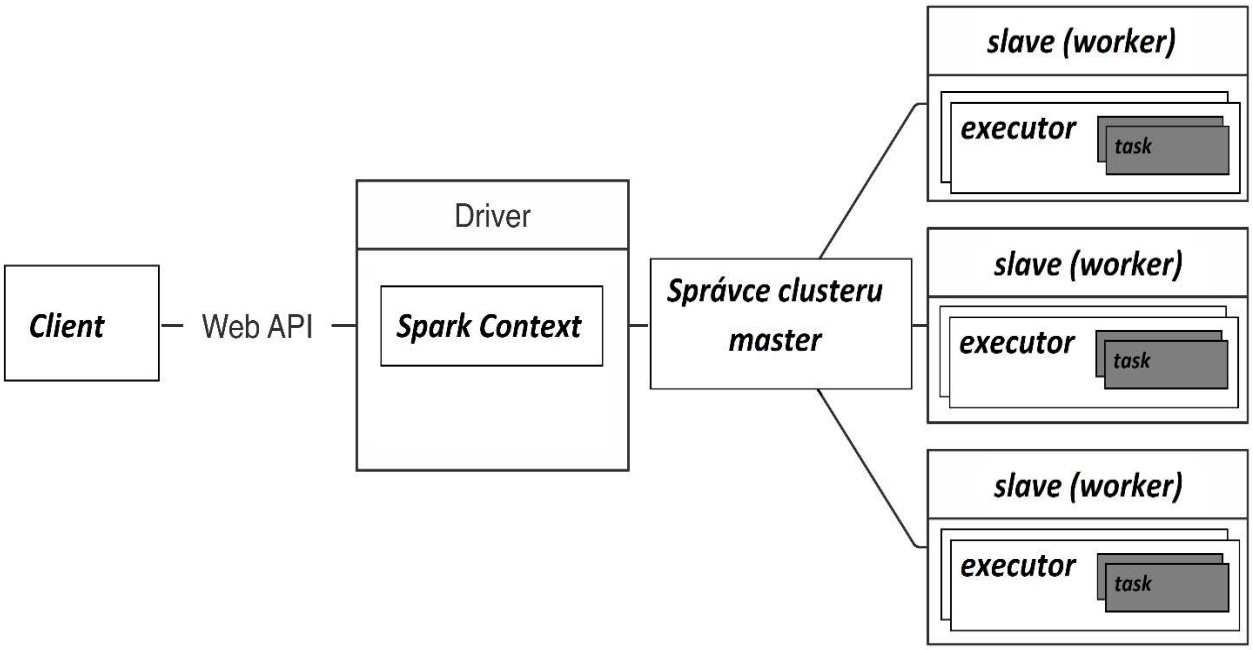


Schéma 8 Architektura clusteru

3.3.5. Tok dat v reálném světě

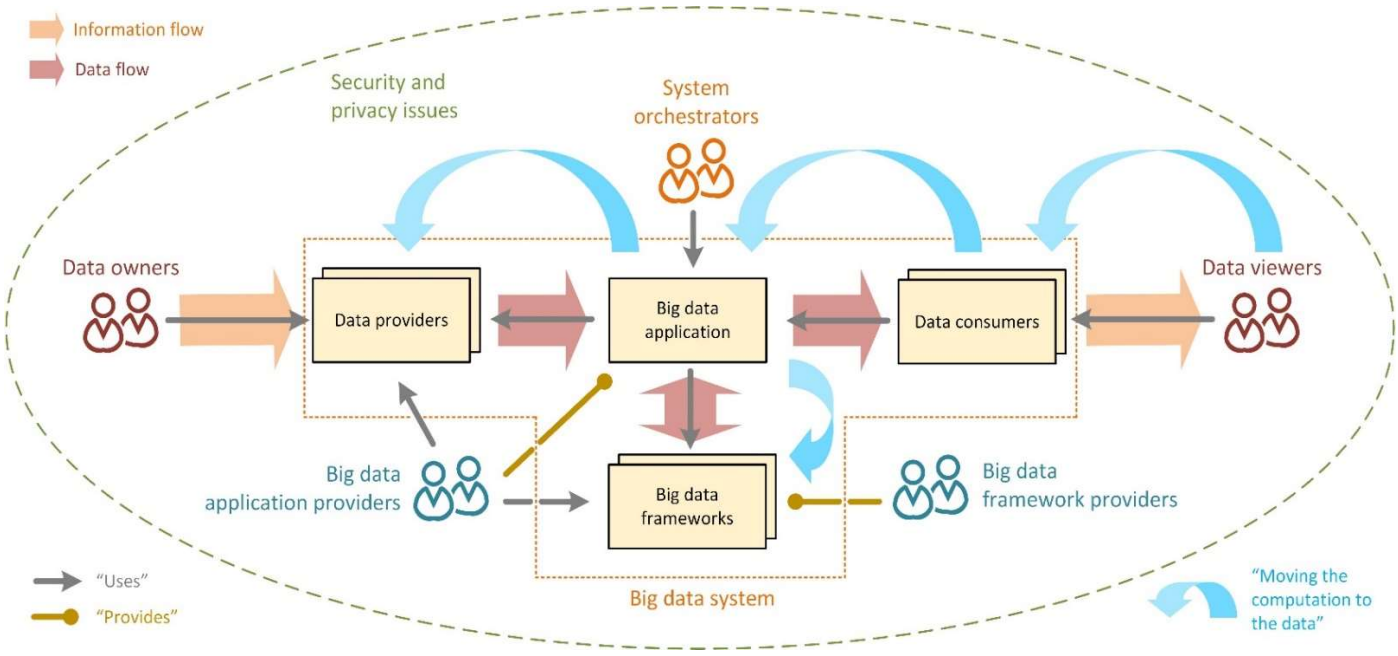


Schéma 9 DataFlow(zdroj: Google pictures)

3.3.6. Průběh chodu aplikace

1) Aplikace se spustí a vytvoří instanci SparkContext/SparkSession (aplikaci lze nazvat Driver (ovladač) pouze, pokud je vytvořen SparkContext/SparkSession).
2) Driver program požádá správce clusteru o prostředky ke spuštění exekutorů.
3) Správce clusteru spustí exekutory.
4) Driver proces probíhá prostřednictvím uživatelské aplikace. V závislosti na akcích a transformacích přes RDD jsou úlohy odesílány exekutorům.
5) Exekutoři spouštějí úkoly a ukládají výsledky.
6) Pokud některý pracovník havaruje, jeho úkoly budou odeslány různým exekutorům, aby je znovu zpracovali.

Chování clusteru v případě chyby

V reálném užití můžeme jako u všeho i zde narazit na selhání systému. Jaké je tedy chování, když je dotaz odeslán do uzlu a uzel se zhroutí těsně před provedením úlohy? Pokud je do clusteru zaveden nový uzel, co detekuje přidání tohoto nového stroje? Přiřadí se novému počítači oddíl, který nebyl zpracován?

Master považuje u pracovníka za selhání, když po dobu 60 sekund neobdržel tzv. heartbeat zprávu (podle spark.worker.timeout). V takovém případě je oddíl přiřazen jinému worker uzlu.

Pokud jsou podřízené jednotky spuštěny, Spark-master již nezjistí přidání nového uzlu do clusteru, protože před odesláním aplikace v clusteru si master přečte soubor s IP adresami všech worker uzlů a spouští instanci slave na každém zadaném počítači. Spark-master po spuštění nepřečte tento konfigurační soubor, takže není možné přidat nový uzel, jakmile je úloha spuštěna.

3.3.7. Využití

Apache Hadoop MapReduce je programovací model pro zpracování velkých souborů dat pomocí paralelního distribuovaného algoritmu. Vývojáři mohou psát masivně paralelizované operátory, aniž by se museli starat o distribuci práce a odolnost proti chybám. Výzvou pro MapReduce je však sekvenční vícekrokový proces, který je zapotřebí ke spuštění úlohy. S každým krokem čte MapReduce data z clusteru, provádí operace a zapisuje výsledky zpět do HDFS. Protože každý krok vyžaduje čtení a zápis z disku, jsou úlohy MapReduce pomalejší kvůli latenci diskových I/O.

Apache Spark byl vytvořen, aby řešil omezení MapReduce tím, že provádí zpracování v paměti, snižuje počet kroků v úloze a znovu používá data v několika paralelních operacích. Se Sparkem je tak potřeba pouze jeden krok, kde jsou data načtena do paměti, provedeny operace a výsledky zapsány zpět – výsledkem je mnohem rychlejší provádění. Apache Spark také znovu používá data pomocí mezipaměti (cache), aby výrazně urychlil algoritmy strojového učení, které opakovaně volají funkci na stejné datové sadě. Opětovné použití dat je dosaženo vytvořením DataFrames, abstrakce přes Resilient Distributed DataSet (RDD), což je kolekce objektů, které jsou uloženy v paměti a znovu

použity v několika operacích Spark. To dramaticky snižuje latenci, díky čemuž je Spark několikrát rychlejší než MapReduce, zejména při strojovém učení a interaktivní analýze (12).

Apache Spark je univerzální systém distribuovaného zpracování používaný pro velké objemy dat. Byl nasazen v každém typu případů použití velkých dat k detekci vzorců a poskytování přehledu v reálném čase. Příklady případů použití zahrnují:

- Finanční služby

Apache Spark se používá v bankovnictví k předpovídání odchodu zákazníků a doporučování nových finančních produktů. V investičním bankovnictví se Spark používá k analýze cen akcií, k předpovídání budoucích trendů.

- Zdravotní péče

Apache Spark se používá k budování komplexní péče o pacienty tím, že zpřístupňuje data předním zdravotnickým pracovníkům pro každou interakci s pacientem. Spark lze také použít k predikci/ doporučení léčby pacienta.

- Výroba

Apache Spark se používá k eliminaci prostojů zařízení připojeného k internetu tím, že doporučuje, kdy provést preventivní údržbu.

- Maloobchod

Apache Spark se používá k přilákání a udržení zákazníků prostřednictvím personalizovaných služeb a nabídek.

3.3.8. API (Application Programming Interface)

API označuje v informatice rozhraní pro programování aplikací. Tento termín používá softwarové inženýrství. Jde o sbírku procedur, funkcí, tříd či protokolů nějaké knihovny (ale třeba i jiného programu nebo jádra operačního systému), které může programátor využívat. API určuje, jakým způsobem jsou funkce knihovny volány ze zdrojového kódu programu. Rozhraní, které se vytváří při kompilaci a je využíváno při běhu programu, se nazývá ABI.

Funkce API jsou programové celky, které programátor používá namísto toho, aby je sám naprogramoval.

Druhy API

1. **RDD (Resilient Distributed DataSet)**

    a. Popis

První API Sparku. Stále používané a nejefektivnější API pro určité operace. DataSety (viz další druhy) jsou postavené na základech RDD, tzn. Spark je na RDD založený. RDD je zjednodušeně sada řádků obsahujících informace. Díky tomu, že jsou data rozdělena do řádků, je lze distribuovat do rozdílných

zařízení, kde mohou být paralelně zpracovávána. Díky této skutečnosti se jedná o „Distributed DataSet“. „Resilient“ je to proto, že Spark se vždy postará, aby zpracování proběhlo, nezávisle na tom, kde se tak děje. Zároveň je schopný sám nahradit „uzel“, pokud při výkonu operace neočekávaně přeruší svou činnost. RDD je schopen funkčního programovaní. To znamená, že do parametru metody lze zadat funkci, např. rdd.map(x =>x*x).

a. RDD SparkContext

První, co je třeba udělat při práci s RDD je vytvoření tzv. SparkContext. Ten zajišťuje, že DRR bude „Resilient & Distributed“. Není tak nutné psát řádky kódu pro zajištění distribuce dat přes celý cluster, protože to dělá Spark sám. Jediné, co je třeba řešit je transformace dat. Zároveň je schopný i vytvořit pro vás objekt, přes který načtete následně data.

Pro vytvoření RDD je zásadní, předat nějaká data.

RDD

Tato ukázka představuje RDD jako způsob uložení výsledků analýzy nejlépe hodnocených filmů.

```
/** Import potřebných knihoven */

import org.apache.spark._
import org.apache.log4j._
import org.apache.spark.rdd.RDD

/** Hledání filmu s nejvíce hodnoceními*/
object PopularMovies {

  /** Main – zde se vykonává příkaz */
  def main(args: Array[String]) {

     // // Vytvoření SparkContextu
// local[*] – využití všech dostupných jader procesoru. Při definování přesného počtu se * nahradí číselnou hodnotou
// “PopularMovies” – název aplikace
    val sc = new SparkContext("local[*]", "PopularMovies")

    // Čtení dokumentu řádek po řádce
    val lines: RDD[String] = sc.textFile("source")

    // Funkce Map vytvoří tuple(dvojici) – (movieID, 1)
    val movies = lines.map(x => (x.split("\t")(1).toInt, 1))

    // Sečtení všech hodnot 1 v tuple pro každý film
    val movieCounts = movies.reduceByKey( (x, y) => x + y )

    // Prohození pořadí ve dvojici – (movieID, count) -> (count, movieID)
    val flipped = movieCounts.map( x => (x._2, x._1) )
```

```
            // Seřazení dat podle prvního sloupce
            val sortedMovies = flipped.sortByKey()

            // Collect („sběr") a tisk výsledků do konzole
            val results = sortedMovies.collect()

            results.foreach(println)
          }
        }
```

2. **DataFrame**
   DataFrame je příkladem modernějšího API, se kterým Spark pracuje. S příchodem Spark verze 2 přišla i možnost efektivního využití SQL, tzv. SparkSQL. DataFrame je svými vlastnostmi přirovnatelný k relační databázi. Skládá se z řádků, které obsahují strukturované informace a má schéma, díky kterému lze informace ukládat efektivněji. Díky těmto vlastnostem lze k DataFramu přistupovat jako k SQL databázi a používat nad ním SQL dotazy. Jistou nevýhodou DataFramu je, že nevíme, jaký typ mají data, dokud ho nedefinujeme.

Tato ukázka představuje DataFrame jako strukturu pro uložení analýzy dat. Aplikace vyhodnocuje ze zadaných dat průměrný počet přátel podle věku.

```
/** Import potřebných knihoven */

import org.apache.spark.sql._
import org.apache.log4j._


object DataFrame_Exercise {

/** Vytvoření CaseClass */
  case class Person(id: Int, name: String, age: Int, friends: Int)

  /** Main – zde se vykonáva příkaz */
  def main(args: Array[String]) {

    // Nastavení loggeru pro výpis do konzole (pouze chyby)
    Logger.getLogger("org").setLevel(Level.ERROR)


/** Vytvoření instance SparkSession*/
    val spark = SparkSession
      .builder()
      .appName("SparkSQL-friends")
      .master("local[*]")
      .getOrCreate()

/** Nastavení čtecího příkazu: jestli má schéma a hlavičku, cesta k souboru, alias*/
    import spark.implicits._
    val people = spark.read
```

```
      .option("header", "true")
      .option("inferSchema", "true")

.csv("E:\\skola\\DP\\Courses_udemy\\SparkScalaCourse\\data\\fakefriends.csv
")
      .as[Person]
/**Výpis schématu do konzole */
    println("This is schema")
    people.printSchema()
/** Výběr sloupců pomocí select*/
    println("Select age and number of friends")
    val ageFriends = people.select("age", "friends")
/**Seřazení dat podle věku a průměrných přátel */
    println("Select avg number of friends for each age")
    ageFriends.groupBy("age").avg("friends").orderBy("age").show()


  }
}
```

3. **DataSet**
   DataSet je velmi podobný DataFramu. Technicky je DataFrame DataSetem obsahujícím pouze řádkové objekty. Rozdílem je, že u DataSetu víme při kompilaci, co který sloupec obsahuje a jaký je jeho typ. Můžeme buď vytvořit tzv. „case class“, která určí název sloupce a jeho typ, nebo můžeme přímo vytvořit DataSet u kterého řekneme, že chceme typ, např. String a Double, na každém řádku. DataSet tedy zná schéma při kompilaci a díky tomu lze odhalit chyby související s datovým typem při psaní „kódu“ a ne až při jeho spuštění. Toto je velkou výhodou při optimalizaci a vývoji programu obecně.

Tato aplikace vyhodnocuje celkovou útratu každého zákazníka v prodejně ze zadaných dat.

```
/**Import potřebných knihhoven */

import org.apache.spark.sql.types.{FloatType, IntegerType, StringType,
StructType}
import org.apache.log4j._
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._

object DataSet_Exercise {

/** Vytvoření CaseClass*/
  case class Customer (customer:String, order:String, cashVal:Float)

/**Main – zde se vykonává příkaz */
  def main(args: Array[String]): Unit = {
```

```
    // Nastavení loggeru na výpis pouze erroru
    Logger.getLogger("org").setLevel(Level.ERROR)

/**Vytvoření instance SparkSession */
    val spark = SparkSession
      .builder()
      .appName("MoneySpent")
      .master("local[*]")
      .getOrCreate()

/** Definice schematu*/
    val orderSchema = new StructType()
      .add("customer", StringType, nullable = true)
      .add("order", StringType, nullable = true)
      .add("cashVal", FloatType, nullable = true)

/** Nastavení čtecího příkazu: jestli má schéma a hlavičku, cesta k souboru, alias*/
    import spark.implicits._
    val orders = spark.read
      .schema(orderSchema)
      .csv("E:\\skola\\DP\\Courses_udemy\\SparkScalaCourse\\data\\customer-
orders.csv")
      .as[Customer]

/**Selekce dat */
    val data = orders.select("customer","cashVal")

/**Agregační funkce nad vybranými daty */
    val dataF = data.groupBy("customer").agg(round(sum("cashVal"), scale =
2).alias("totalSpent"))

/** Seřazení dat*/
    val dataS = dataF.sort("totalSpent")

/**Kolekce dat */
    val res = dataS.collect()

/**Výpis dat do konzole */
    for (result<-res) {
      val customer = result(0)
      val cash = result(1)
      println(s"Customer n.:$customer spent total of: $cash USD")
    }
  }
}
```

4. Praktická část

Obsahem této kapitoly jsou ukázky praktického využití Pythonu společně s Apache Spark v oblasti geodézie. Jsou v ní popsána data, na kterých byly ukázky prováděny, a poté i samotné aplikace s popisem kódu. Ten vnáší pohled na problematiku zpracování prostorových dat pomocí již zmíněných nástrojů.

4.1. Data

4.1.1. Zdroj dat + kdo je poskytl

Původním plánem bylo využít souborů dat formátu Rinex[5]. Tento formát však neobsahuje spočítané hodnoty, ale pouze data k jejich spočítání, se kterými se až poté pracuje. Vzhledem k nedostatku času na napsání programu ke čtení a výpočtu či vložení dat formátu Rinex do jiné, již vytvořené aplikace, jsem byl nucen najít data jinde. Onou záchranou se mi stal Ing. Zdeněk Vyskočil, Ph.D, který mi poskytl data měřená přímo na půdě fakulty.

Jedná se o RTK pozici měřenou pomocí GNSS přijímače UBLOX-ZED9N. Měření probíhá na střeše budovy B Fakulty stavební ČVUT. Měření probíhá v nepravidelných intervalech a používají se různé kombinace satelitních systémů.

Data jsou uložena na Raspberry Pi4, což je malý jednodeskový počítač s velikostí platební karty. Dále jsou data v různých intervalech exportována do formátu *.csv* a uložena na virtuální server pronajímaný u společnosti Contabo. K tomuto serveru je přístup pomocí webové aplikace či pomocí SSH. SSH nebo Secure Shell je síťový komunikační protokol, který umožňuje dvěma počítačům komunikovat a sdílet data.

4.1.2. Formát/ struktura dat

Data jsou uložena na MYSQL serveru ve formátu .csv. Jsou rozdělena do více souborů podle data jejich akvizice (měření). Jde tedy o textový formát, který užívá čárku jako oddělovač. Data obsahují hlavičku, tzn. header, ve kterém je popsán obsah řádků (jednotlivé sloupce) tak, jak jdou po sobě. Jednotlivé informace jsou samy o sobě v textovém formátu (String). Je tedy potřeba je pro početní úlohy přeformátovat. O tom však více až v dalších kapitolách.

4.1.3. Ukázka obsahu dat

Zde vidíme hlavičku každého souboru. V každém souboru se tedy vyskytují informace o datu a čase měření, naměřené zeměpisné šířce a délce, naměřené výšce, počtu viditelných a „použitých“ družic atd.

```
start_time,data_index,config,lat_mean,lat_std,lon_mean,lon_std,alt_mean,alt_std,hdop_mean,fixing_time,GP_used_mean,GA_used_mean,GL_used_mean,GP_visible_mean,GA_visible_mean,GL_visible_mean,used_count,visible_count
```

[5] Popis formátu rinex k dispozici na http://acc.igs.org/misc/rinex304.pdf

Data jsou vypsána v řádku za sebou, kde oddělovačem je čárka. Jde o textový formát.

2021-11-05 11:01:16,0,GAGL,50.10389768733333,1.6802777458387252e-08,14.387835666833334,1.1388346762580303e-08,260.8912000000001,0.0067052218457016,0.756,44.0,0,6,7,0,9,9,13,18

4.2. Práce s daty

4.2.1. Připojení k datům/ načtení dat

Pro získání dat mi Ing. Zdeněk Vyskočil, Ph.D., poskytl přístupové údaje a heslo k MYSQL serveru, na kterém jsou data uložena. Jak již bylo zmíněno, data jsou uložena v databázi a přistupuje se k nim přes SQL dotazy. Je tedy třeba otevřít SSH tunel ze zařízení na server. Pro účely této práce byly vyzkoušeny dvě možnosti, z nichž jedna byla rovnou zhodnocena jako neefektivní a nevhodná, a proto se o ní zmíním pouze krátce bez jakýchkoliv ukázek.

První možností připojení a získání dat je přes tzv. windows console. Toto řešení však není automatizované a vyžaduje ruční psaní příkazů pro každé jedno připojení. Rovněž data je třeba stáhnout, uložit a pak znova načítat do programu, což není ideální. Bylo tedy třeba najít jiné, optimálnější řešení – viz druhá možnost.

Druhou možností je připojení na server přímo z prostředí aplikace. Pro potřeby připojení nebylo užito Apache Spark, ale knihoven Pandas, PyMySQL a sshtunnel vytvořených pro programovací jazyk Python.

Pandas je knihovna vytvořená Wesem McKinneym pro programovací jazyk Python pro analýzu dat. Je postavena na dvou základních Pythonových knihovnách – Matplotlib pro vizualizaci dat a NumPy pro matematické operace. Pandas funguje jako obal nad těmito knihovnami, což vám umožňuje přistupovat k mnoha metodám Matplotlib a NumPy s menším množstvím kódu. Například .plot() v Pandas kombinuje několik metod Matplotlib do jediné metody a umožňuje tak vykreslit graf v několika řádcích.

Před Pandas používala většina analytiků pro získávání a přípravu dat Python a poté po zbytek svého pracovního postupu přešli na jazyk více specifický pro doménu, jako je R. Pandas představil dva nové typy objektů pro ukládání dat, které usnadňují analytické úlohy a eliminují potřebu přepínat nástroje: Series, které mají strukturu podobnou seznamu, a DataFrames, které mají tabulkovou strukturu. Díky využití DataFrameu v knihovně Pandas i v Apache Spark lze použít čtení SQL dotazu na server přes Pandas, uložit data jako Pandas DataFrame a později je převést na Spark DataFrame. Tento postup si ukážeme v následujících ukázkách.

Připojení na vzdálený server

Pro efektivnější užívaní a psaní „kódu“ lze vytvořit modul, který lze stejně jako knihovnu importovat a poté využít jeho předdefinované funkce bez nutnosti je znova psát. Pro potřeby připojení byl tedy vytvořen pythonovský modul pracovně pojmenovaný SSH_connection.py, který definuje funkce potřebné pro navázaní spojení se serverem a následný výběr a uložení dat do proměnných.

V následující kapitole si ukážeme zdrojový kód tohoto modulu.

Modul SSH_connection.py

Stejně jako v každé aplikaci je nejprve třeba importovat využité knihovny. V případě tohoto modulu je nejdůležitější součástí knihovna Pandas, o které jsem se již krátce zmínil výše. Neznamená to však, že by modul fungoval pouze s ní. Pro potřeby připojení a výběru dat bylo zapotřebí importovat i knihovny PyMySQL, logging a sshtunnel.

```
import pandas as pd
import pymysql
import logging
import sshtunnel
from sshtunnel import SSHTunnelForwarder
```

Následně je třeba zadat přístupové údaje. Je třeba definovat adresu serveru, uživatelské jméno (username) pro přístup na server a heslo, uživatelské jméno v databázi, její název a heslo k ní. Local host je potřeba uvést vždy a podle dostupných zdrojů se uvádí přímo řetězec „Local host“ anebo adresa „127.0.0.1“.

```
# Configure your credentials
ssh_host = „IP adresa serveru“
ssh_username = 'Username'
ssh_password = 'Heslo k přístupu na server'
database_username = 'Uživatelské jméno v databázi'
database_password = 'Heslo pro přístup k databázi'
database_name = 'Název databáze'
localhost = '127.0.0.1'
```

Po zadání přístupových údajů přicházejí na řadu samotné definice funkcí, užitých pro připojení a selekci dat.

Funkce *open_ssh_tunnel* využíva knihovny sshtunel. Knihovna sshtunnel jak již název napovídá, umožňuje otevření ssh tunelu ke vzdálenému stroji.

Vstupními hodnotami je adresa serveru, název a heslo. Funkce žádné hodnoty nevrací, pouze otevírá cestu ke vzdálenému stroji a drží ji otevřenou, dokud ji „manuálně“ nezavřeme.

Rovněž zde definujeme, co vše se bude vypisovat do konzole.

```
def open_ssh_tunnel(verbose=False):
    """Open an SSH tunnel and connect using a username and password.

    :param verbose: Set to True to show logging
    :return tunnel: Global SSH tunnel connection
    """

    if verbose:
        sshtunnel.DEFAULT_LOGLEVEL = logging.DEBUG
```

```
    global tunnel
    tunnel = SSHTunnelForwarder(
        (ssh_host, 22),
        ssh_username=ssh_username,
        ssh_password=ssh_password,
        remote_bind_address=('127.0.0.1', 3306)
    )

    tunnel.start()
```

Pro funkci *mysql connect* je hlavní využívanou knihovnou PyMySQL, což je knihovna pro Python umožňující připojení na MYQL server přes SSH tunel a stabilizaci připojení na SQL databázi, která běží na serveru (13).

Vstupními hodnotami jsou local host, název databáze, uživatelské jméno a heslo. Funkce opět žádné hodnoty nevrací, pouze vytváří připojení na požadovanou databázi na serveru, dokud jej stejně jako SSH tunel neukončíme.

```
def mysql_connect():
    """Connect to a MySQL server using the SSH tunnel connection

    :return connection: Global MySQL database connection
    """

    global connection

    connection = pymysql.connect(
        host='127.0.0.1',
        user=database_username,
        passwd=database_password,
        db=database_name,
        port=tunnel.local_bind_port
    )
```

Funkce *run_query* umožňuje zadat SQL dotaz na data na připojené databázi.

Funkce vrací Pandas DataFrame. Důvodem pro využití Pandas je to, že jsem v dokumentaci Sparku nenašel funkci, která by dokázala vznést SQL dotaz na vzdálenou databázi, připojenou přes SSH.
Z tohoto důvodu je v této práci využito knihovny Pandas.

Vstupem do funkce knihovny Pandas *read_sql_querry* je samotný SQL dotaz, který se zadává při volání funkce a již etablované připojení na SQL databázi.

Funkce vrací Pandas DataFrame, který je velmi podobný Spark DataFrame.

```
def run_query(sql):
    """Runs a given SQL query via the global database connection.

    :param sql: MySQL query
```

```
    :return: Pandas dataframe containing results
    """

    return pd.read_sql_query(sql, connection)
```

Pro ukončení připojení k SQL databázi je definována funkce *mysql_disconnect.* Nad proměnnou *connection* se užije jednoduchého příkazu *.close().*

```
def mysql_disconnect():
    """Closes the MySQL database connection.
    """

    connection.close()
```

Podobně jako u předchozí funkce je i zde, ve funkci *close_ssh_tunel,* využito nad proměnnou příkazu *.close().*

```
def close_ssh_tunnel():
    """Closes the SSH tunnel connection.
    """

    tunnel.close
```

Nyní máme připravený modul a funkce pro připojení na MySQL databázi běžící na vzdáleném serveru. Můžeme se tedy připojit na vzdálenou databázi a pomocí SQL dotazů z ní přebírat data a s těmi dále pracovat. DataFrame získaný touto cestou však není způsobilý pro zpracování pomocí Apache Spark, respektive PySpark. Je proto třeba provést jisté úkony, než s takto získanými daty začneme provádět další kroky jejich zpracování a získávání informací.

4.2.2. Úprava dat

Začátkem je opět import použitých modulů a knihoven. SSH_connection je modul, jehož tvorba a součásti byly popsány v předchozím textu. Díky tomuto modulu lze využívat veškeré funkce, které obsahuje.

SparkSession je vstupní bod k programování Spark pomocí rozhraní Dataset a DataFrame API. SparkSession lze použít k vytvoření DataFrame, registraci DataFrame jako tabulek, spouštění SQL přes tabulky, mezipaměti tabulek a čtení parketových souborů. (5)

Pyspark.sql.types je knihovna obsahující všechny typy podporované Sparkem. Díky tomu je lze vytvářet, přetypovávat data a obecně s nimi pracovat.

Pyspark.sql.functions je část knihovny obsahující různé funkce, které lze provádět nad různými datovými typy. Příkladem užitým v dalším kódu je například funkce *col()*, matematické funkce *sqrt()* nebo *abs(),* či sortovací funkce *asc(), desc().*

```
from SSH_connection import *
from pyspark.sql import SparkSession
from pyspark.sql.types import *
from pyspark.sql.functions import *
```

Základem každé aplikace v prostředí ApacheSpark je SparkSession. Dříve byl místo SparkSession využíván SparkConf. Ten slouží ke konfiguraci pro aplikaci Spark a používá se k nastavení různých parametrů Spark jako párů klíč-hodnota (5). Byl však nahrazen modernějším a všestrannějším SparkSession.

SparkSession má různé atributy. Atribut třídy *builder* má na starosti vytváření instancí SparkSession. Master je atribut určující počet jader procesorů, jež budou pro danou úlohu využity. [*] v tomto případě značí všechna dostupná jádra. Atribut *appName* slouží k pojmenování instance (aplikace).

```
sc = SparkSession.builder \
    .master("local[*]") \
    .appName("Import") \
    .getOrCreate()
```

Zde přichází na řadu funkce vytvořené v modulu *SSH_connection*. *Open_ssh_tunnel a mysql_connect* slouží k připojení na vzdálenou MySQL databázi pomocí ssh tunelu. Funkce *run_query* na předem definovaném připojení spouští zde zadaný SQL dotaz na selekci dat a následně je ukládá do proměnné *df* v podobě Pandas DataFramu. Příkaz *.head()* se používá pro získání prvních n řádků. Při užití bez n vybírá vše. *Mysql_disconnect* a *close_ssh_tunnel* slouží k uzavření připojení ke vzdálené MySQL databázi.

V případě, že by bylo třeba získávat data kontinuálně či postupně, by tyto poslední dva příkazy byly použity až na úplném konci kódu. Jelikož pro ukázku a testování prostředí Apache Spark respektive PySpark „saháme“ pro data pouze jednou, je spojení terminováno ihned po získání dat.

```
open_ssh_tunnel()
mysql_connect()
df = run_query("SELECT * FROM ublox_test4")
df.head()
mysql_disconnect()
close_ssh_tunnel()
```

Pokud by byl požadavek automatizovaně ukládat data na lokální disk, lze využít příkazu *.to_excel()*. Tento příkaz se spouští nad DataFramem a je třeba mu definovat jeden povinný parametr a tím je cesta k uložení souboru na disk (tzn. kam chceme soubor uložit). Lze rovněž definovat i název listu excelu (sheet :*sheet_name*) či formát uložení dat *na_rep*, který je v tomto případě uveden jen pro ukázku a nechává původní.

Jelikož typ dat Pandas *Timestamp* není kompatibilní s formátem stejného názvu v Apache Spark, bylo třeba data ve sloupci “start_time“ přetypovat na typ *String* neboli *textový řetězec*. Je tak učiněno

pomocí příkazu *.astype()* který se definuje nad sloupcem Dataframu a jako proměnná se definuje typ, na který chceme data přeformátovat.

Funkce *print(df.dftypes)* je zde užita pro zobrazení a kontrolu typů dat, které DataFrame obsahuje.

```
# df.to_excel(r'E:\skola\DP\DP\Outputs_write\Output_mysql.xlsx',
sheet_name='Sheet1', na_rep="")

df['start_time'] = df['start_time'].astype('str')
print(df.dtypes)
```

Jejím výstupem, viz níže, je seznam typů podle sloupců. Sloupec "start_time" je uveden jako *object*, což v Pandas znamená, že obsahuje *String, unicode nebo mixed types* (tzn. text nebo „mixované" numerické a nenumerické hodnoty).

```
start_time       object
data_index        int64
config           object
lat_mean        float64
lat_std         float64
……..
```

Dále je třeba definovat schéma DataFramu pro jeho převod z Pandas DataFrame do Spark DataFrame. To je uložené do proměnné s názvem *schema.*

Pomocí funkce *StructType* je možné definovat každý řádek a jeho datový typ. Zde je třeba si dát pozor na to, aby datové typy definované zde korelovaly s datovými typy, které obsahuje Pandas DataFrame. V případě, že by zde byl požadavek na jiný datový typ, než je ten původní, by bylo třeba provést přetypování. Přetypování lze provést již v Pandas DataFramu, jako u sloupce "start_time", kde to bylo nutné, nebo pak až nad Spark DataFramem.

```
schema = StructType([
    StructField("start_time", StringType()),
    StructField("data_index", IntegerType()),
    StructField("config", StringType()),
    StructField("lat_mean", DoubleType()),
    StructField("lat_std", DoubleType()),
    StructField("lon_mean", DoubleType()),
    StructField("lon_std", DoubleType()),
    StructField("alt_mean", DoubleType()),
    StructField("alt_std", DoubleType()),
    StructField("hdop_mean", DoubleType()),
    StructField("fixing_time", IntegerType()),
    StructField("GP_used_mean", IntegerType()),
    StructField("GA_used_mean", IntegerType()),
    StructField("GL_used_mean", IntegerType()),
    StructField("GP_visible_mean", IntegerType()),
    StructField("GA_visible_mean", IntegerType()),
    StructField("GL_visible_mean", IntegerType()),
    StructField("used_count", IntegerType()),
```

```
    StructField("visible_count", IntegerType())
])
```

Příkaz *sc.conf.set("spark.sql.execution.arrow.enabled", "true")* spouští, aktivuje či umožňuje použití formátu Apache Arrow.

Apache Arrow je sloupcový datový formát v paměti, který se ve Sparku používá k efektivnímu přenosu dat mezi procesy JVM a Python. To je v současnosti nejvýhodnější pro uživatele Pythonu, kteří pracují s daty Pandas/NumPy. Jeho použití není automatické a může vyžadovat drobné změny v konfiguraci nebo kódu, aby bylo možné plně využít výhod a zajistit kompatibilitu. (5)

V této chvíli si nemám, jak ověřit jeho nutnost použití v kódu, protože jeho aktivace byla při vývoji jedním z pokusů, jak zprovoznit konverzi mezi Pandas DataFrame a Spark DataFrame. Je tedy možné, že aktivovat použití Apache Arrow není potřeba, ale jelikož jde o proces v pozadí příkazu *.createDataFrame()* a rovněž podle popisu v oficiální dokumentaci usuzuji, že stojí za konverzí formátu a jeho aktivace v kódu není nijak omezující.

Příkaz *.createDataFrame()* slouží k vytvoření Spark DataFramu a je třeba mu definovat vstupní hodnoty, v tomto případě Pandas DataFrame a schéma dat.

```
sc.conf.set("spark.sql.execution.arrow.enabled", "true")
spark_DF = sc.createDataFrame(df, schema=schema)
```

Pro konverzi sloupce “start_time“ z pomocného formátu *String* na formát *TimeStamp* je použito příkazu *.withColumn().* Příkaz *.withColumn()* má dva parametry. Prvním je název sloupce, se kterým pracujeme, a druhý je samotný příkaz, kde definujeme, co do sloupce vkládáme. Pomocí tohoto příkazu lze sloupce i vytvářet, kde prvním parametrem je pak název nového sloupce a druhým je hodnota vkládaná do sloupce. Hodnotou může být jiný sloupec, konstanta, či hodnota vypočítaná z ostatních sloupců či proměnných uložených mimo DataFrame.

```
spark_DF = spark_DF.withColumn("start_time",
col("start_time").cast(TimestampType()))

spark_DF.printSchema()
spark_DF.select("start_time").show()
```

Pro kontrolu je použita funkce *.printSchema()*, které vypíše datové typy v jednotlivých sloupcích a příkaz *.select()*, který v tomto případě vybírá sloupec “start_time“ a pomocí příkazu *.show()* jej zobrazí do konzole.

Výpisy v terminálu pak vypadají takto:

```
root
 |-- start_time: timestamp (nullable = true)
 |-- data_index: integer (nullable = true)
 |-- config: String (nullable = true)
 |-- lat_mean: double (nullable = true)
```

```
 |-- lat_std: double (nullable = true)
……
+------------------+

|        start_time|
+------------------+
|2021-11-05 11:01:16|
|2021-11-05 11:08:31|
|2021-11-05 11:18:30|
|2021-11-05 11:19:52|
|2021-11-05 11:24:05|
|2021-11-05 11:59:03|
|2021-11-05 12:09:34|
|2021-11-05 12:16:46|
……
```

4.3. Ukázky jednoduchých geodetických aplikací

4.3.1. 2D Transformace

Základní operací pro práci se souřadnicemi je jejich transformace do jiných souřadnicových systémů. Vytvořil jsem proto ukázkovou aplikaci, která nahraje soubor z lokálního disku, na kterém jsou měřená data, a přetransformuje je do zvoleného souřadnicového systému. Následně výsledné souřadnice vypíše do terminálu a uloží do souboru formátu .csv.

V ukázce byl využit soubor s několika tisíci měřeními. V reálném použití však může být počet měření mnohonásobně vyšší. Aplikaci lze použít pro transformace souřadnic v různých souřadnicových systémech, pokud jsou dostupné v knihovně PyPROJ.

Základem je opět import využitých knihoven. Využité knihovny jsou téměř totožné s knihovnami zmíněnými v předchozí kapitole, proto ty již popsané uvádím pouze výčtem. Jedná se tedy o pyspark.sql, který importuje SparkSession, types a functions.

Novou knihovnou je tu však knihovna PyPROJ. PyPROJ je rozhraní Pythonu k PROJ, což je knihovna kartografických projekcí a transformací souřadnic. PROJ provádí kartografické transformace. Převádí ze zeměpisné délky, šířky do nativních souřadnic x, y projekce mapy, a naopak pomocí PROJ. (14)

```
from pyspark.sql import SparkSession
from pyspark.sql.types import *
from pyspark.sql.functions import *
from pyproj import *
```

Základem každé aplikace v prostředí Apache Spark je SparkSession. Zde opět využijeme povinný atribut *builder, který* má na starosti vytváření instancí SparkSession. Jelikož je požadavek na využití veškerého možného výkonu, nastavíme atribut *master* na [*]. Atribut *appName* slouží k pojmenování instance (aplikace).

```
sc = SparkSession.builder \
    .master("local[*]") \
    .appName("Transformace_souradnic") \
    .getOrCreate()
```

Pro testování bylo využito souboru na disku. V následujících aplikacích tedy z důvodu úspory času načítaní již není užito připojení přes SSH tunel na vzdálenou databázi. Do proměnné *path* je vložena cesta k souboru ve formátu *String*. Je nutné uvést kompletní cestu i s informací o formátu souboru.

K načtení dat ze souboru se používá příkaz *.read.* Tento příkaz se spouští nad instancí SparkSession a je třeba mu dát i formát, který má načítat. Zde je načítán *.csv* a proto má příkaz *.csv*.

Vstupem do funkce je soubor v námi předem definované cestě. Lze ji definovat i přímo v příkazu. Dalšími argumenty jsou *sep=‘‘* a *header=TRUE/FALSE)*. *Sep* udává typ separátoru. V tomto případě je jím čárka. *Header* definuje přítomnost/ absenci záhlaví, tzn. prvního řádku definujícího názvy sloupců.

Příkaz *.printSchema()* je zde užit pro kontrolu načtení dat a jejich typů.

```
path = 'E:\skola\DP\logovani_ublox\data_bez_cisel\proc_checked.csv'

coords = sc.read.csv(path, sep=',', header=True)
coords.printSchema()
```

Výstup z *.printSchema()*, viz. níže, ukazuje, že všechna data jsou ve formátu *String*. Je tedy opět potřeba provést přetypování.

```
root
 |-- start_time: String (nullable = true)
 |-- data_index: String (nullable = true)
 |-- config: String (nullable = true)
 |-- lat_mean: String (nullable = true)
……..
```

Změna typu formátu sloupce se provede pouze u vybraných sloupců. Použije se pro to příkaz *.withColumn()*. Prvním argumentem je název sloupce, nad kterým budeme operace provádět, a druhým argumentem je samotná operace. Zde se využívá příkazu *.cast()*, který slouží ke změně formátu. Příkaz *.cast()* však musí být zavolán na sloupcem, který nelze pouze označit názvem, ale musí být přímo definován jako sloupec pomocí funkce *col()*. Alternativou pro tento příkaz je *coords.start_time.cast()*. Jde pouze o jiný zápis definice sloupce, nad kterým budeme příkaz provádět. *Coords* v našem případě je název DataFramu, ze kterého vybíráme sloupec *start_time*.

```
coords = coords \
    .withColumn("start_time", col("start_time").cast(TimestampType())) \
    .withColumn("lat_mean", col("lat_mean").cast(DoubleType())) \
    .withColumn("lat_std", col("lat_std").cast(DoubleType())) \
    .withColumn("lon_mean", col("lon_mean").cast(DoubleType())) \
    .withColumn("lon_std", col("lon_std").cast(DoubleType())) \
```

```
    .withColumn("alt_mean", col("alt_mean").cast(DoubleType())) \
    .withColumn("alt_std", col("alt_std").cast(DoubleType()))
```

Jelikož jde o úlohu 2D transformace, je použito příkazu *.select()* pro vybrání času měření, zeměpisné šířky a zeměpisné délky. Úloze 3D transformace se budeme krátce věnovat v další kapitole, jelikož jde jen o malé úpravy transformace dvojrozměrné.

```
coords_red = coords.select("start_time", "lat_mean", "lon_mean")
```

Pro jednodušší práci se sloupci byly sloupce přejmenovány na jednoslovné názvy. Do proměnné *newColumns* byly vloženy ve formátu *String* nové názvy sloupců. K přejmenování by se dalo použít i příkazu *.withColumn()*, ale zde je jednodušší užít funkce *.toDF()*, která vytváří nový DataFrame z původního, a jejím argumentem je tento náš vytvořený „header".

```
newColumns = ["start_time", "lat", "lon"]
coords_ren = coords_red.toDF(*newColumns)
```

Nyní již přichází na řadu operace umožňující či související přímo s naší transformací. První, co je třeba definovat, jsou vstupní a výstupní souřadnicový systém. Pro jejich definici se použije funkce *CRS()*, kde argumentem je název, kód či obojí daného souřadnicového systému. Tyto systémy by se daly definovat až při volání samotné transformace. Pro zpřehlednění a zjednodušení kódu jsou však definovány do samostatných proměnných. Transformaci tedy provádíme ze systému WGS84 do systému S-JTSK.

```
# definice systemu
crs_4326 = CRS("WGS84")
crs_sjtsk = CRS("EPSG:5514")
```

Jako další je třeba definovat funkci Transformer. Nad tou využijeme příkazu *.from_crs()*, ve kterém definujeme dva argumenty. Těmi argumenty je vstupní a výstupní systém.

```
transformer = Transformer.from_crs(crs_4326, crs_sjtsk)
```

Jak již bylo zmíněno, všechny předešlé operace by se daly shrnout do jednoho jednořádkového příkazu, který by vypadal viz níže. Pro zpřehlednění kódu však byl tento příkaz rozdělen. Do příkazu *.transform()* vstupují zde přímo definované souřadnice.

```
Transformer.from_crs(CRS("WGS84"),
CRS("EPSG:5514")).transform(49.452263035996815, 12.806988096043456)
```

Pro transformaci souřadnic použijeme RDD transformaci *map().* Map se používá k aplikaci transformační funkce (lambda) na každý prvek RDD/DataFrame a vrací nový RDD. Transformace RDD *map()* se používá k aplikaci jakýchkoli složitých operací, jako je přidání sloupce, aktualizace sloupce, transformace dat atd. Výstup transformací map má vždy stejný počet záznamů jako vstup. DataFrame nemůže použít transformaci *map()*, proto je třeba ho nejprve převést na RDD. (15)

Thákurova 7, 166 29 Praha 6

Zde tedy provádíme samotnou transformaci. Prvním sloupcem v našem výstupu bude sloupec *"start_time"*, který se nemění. Druhým sloupcem budou transformované souřadnice. Toho dosáhneme pomocí definované proměnné transfomer. Ta nám definuje vstupní a výstupní souřadnicový systém. Nad tou pomocí příkazu *.transform()* provedeme transformaci. Argumenty tohoto příkazu jsou sloupce obsahující zeměpisnou šířku a zeměpisnou délku.

```
coords_jtsk_rdd = coords_ren.rdd.map(lambda x:
                                     (x.start_time,
                                      transformer.transform(x.lat, x.lon))
                                     )
```

Pro převedení zpět na DataFrame použijeme příkaz *.toDF()*, v jehož argumentu vypíšeme názvy sloupců.

```
coords_jtsk = coords_jtsk_rdd.toDF(["start_time", "coords"])

coords_jtsk.show()
```

Pro vypsání do terminálu použijeme příkaz *.show().* Výstup viz níže.

```
+-------------------+--------------------+
|       start_time|              coords|
+-------------------+--------------------+
|2021-11-05 11:01:16|[-744928.64612842...|
|2021-11-05 11:08:31|[-744928.65397597...|
|2021-11-05 11:18:30|[-744928.65520504...|
……….
```

Problémem však je, že transformace vrací souřadnice ve formátu *tuple(y,x)*. Pro další práci se souřadnicemi je tedy třeba je rozdělit na dva samostatné sloupce. K tomu opět použijeme transformaci RDD *map.* Prvním sloupcem bude opět "start_time" a do druhého a třetího sloupce vložíme souřadnice y, x. K tomu použijeme indexování, kde index 0 značí první prvek a index 1 značí prvek druhý.

```
coords_sjtsk_rdd = coords_jtsk.rdd.map(lambda x:
                                       (x.start_time, x.coords[0], x.coords[1])
                                       )
```

Pro převedení zpět na DataFrame použijeme opět příkaz *.toDF()* ve kterém definujeme názvy nových sloupců.

```
coords_sjtsk = coords_sjtsk_rdd.toDF(["start_time", "Y", "X"])
```

Výsledný výstup transformovaných souřadnic vypsaný do terminálu:

```
+-------------------+-----------------+------------------+
|       start_time|              Y|                 X|
+-------------------+-----------------+------------------+
|2021-11-05 11:01:16| -744928.646128425|-1040875.8948964932|
|2021-11-05 11:08:31|-744928.6539759758|-1040875.9132928756|
|2021-11-05 11:18:30|-744928.6552050484| -1040875.895084205|
|2021-11-05 11:19:52| -744928.645094439|-1040875.9104210329|
|2021-11-05 11:24:05|-744928.6529076607|-1040875.9120930627|
……..
```

4.3.2. 3D transformace

Postup při provádění trojrozměrné transformace je totožný s postupem u transformace dvojrozměrné až do bodu definice *transformeru*. Při jeho zadávání je potřeba na argumenty vstupního a výstupního systému použít příkaz *.to_3d().* Ten v podstatě předává informaci, že pracujeme v trojrozměrném prostoru a budeme transformovat i výšku.

```
transformer_3d = Transformer.from_crs(
    crs_4326.to_3d(),
    crs_sjtsk.to_3d(),
)
```

Prvním sloupcem v našem výstupu je opět sloupec *"start_time"*, který se nemění. Druhým sloupcem jsou transformované souřadnice spolu s výškou. Toho dosáhneme pomocí definované proměnné transfomer. Ta nám definuje vstupní a výstupní souřadnicový systém. Nad tou pomocí příkazu *.transform()* provedeme transformaci. Argumenty tohoto příkazu jsou sloupce obsahující zeměpisnou šířku, zeměpisnou délku a výšku.

```
coords_jtsk_rdd = coords_ren.rdd.map(lambda x:
                                     (x.start_time,
transformer_3d.transform(x.lat, x.lon, x.alt))
                                     )
```

Souřadnice je poté pro další práci opět třeba rozdělit na samostatné sloupce. Použijeme proto opět transformaci pomocí *RDD map*. Rozdílem je, že oproti transformaci dvourozměrné obsahuje sloupec i třetí hodnotu, a tou je výška.

```
coords_sjtsk_rdd = coords_jtsk.rdd.map(lambda x:
                                       (x.start_time, x.coords[0],
x.coords[1], x.coords[2]))
```

Převod na DataFrame je proveden pomocí příkazu *.toDF(),* jehož argumenty jsou názvy sloupců.

```
coords_sjtsk = coords_sjtsk_rdd.toDF(["start_time", "Y", "X", "alt"])
```

Výsledný výstup transformovaných souřadnic vypsaný do terminálu:

```
+-------------------+-----------------+-----------------+-----------------+
|       start_time|              Y|              X|              alt|
+-------------------+-----------------+-----------------+-----------------+
|2021-11-05 11:01:16|-744928.6489491767|-1040875.8978049168| 215.6483591189608|
|2021-11-05 11:08:31|-744928.6567968796|-1040875.9162014555|  215.662458778359|
|2021-11-05 11:18:30|-744928.6580258079|-1040875.8979926368|215.64915901608765|
|2021-11-05 11:19:52| -744928.647915236|-1040875.9133295111|215.65265893936157|
|2021-11-05 11:24:05|-744928.6557284214|-1040875.9150014935|215.64925885014236|
.....
```

4.3.3. Výpočet vzdálenosti od průměru

Pro účely geodézie jsou téměř vždy měřeny hodnoty vícekrát, aby se co nejvíce zamezilo chybě v měření. K dispozici máme data ze stabilizovaného bodu na střeše fakulty. V ideálním případě, by tedy měly být všechny naměřené hodnoty souřadnic totožné. Ve skutečnosti tomu tak není. V této ukázce je příklad nalezení odlehlých (špatných) měření.

K nalezení odlehlých měření je v této úloze použito vzdálenosti od průměru. Z lokálního disku je načten soubor s daty o měření. Pro každou souřadnici je poté vypočten průměr ze všech měření a od tohoto průměru je vypočtena vzdálenost ke každému bodu. Následně jsou data seřazena podle velikosti odchylek od největší po nejmenší.

Jak již bylo zmíněno, nejprve je třeba naimportovat využívané knihovny. Stejně jako u připojení přes SSH tunel zde budeme využívat knihovnu *Pandas*. Novou využívanou knihovnou je knihovna *Matplotlib*. Matplotlib je komplexní knihovna pro vytváření statických, animovaných a interaktivních vizualizací v Pythonu. Dokáže vytvářet grafické vizualizace od základních sloupcových grafů, přes grafy matic a polí, statistických grafů až po zobrazení vrstevnic. (16)

```
from pyspark.sql import SparkSession
from pyspark.sql.types import *
from pyspark.sql.functions import *
import pandas as pd
import matplotlib.pyplot as plt
```

Základem každé Spark aplikace je vytvoření instance SparkSession. *Master* má argument *local[*]*, což značí, že pracujeme na lokálním stroji a využíváme všechny jeho prostředky.

Cesta k souboru je uložena do proměnné path a k načtení souboru je užito příkazu *.read.csv()*. Oddělovačem je čárka a testovací soubor obsahuje hlavičku, proto je argument *header* nastaven na *TRUE*.

```
sc = SparkSession.builder \
    .master("local[*]") \
    .appName("Vypocet_odchylek") \
    .getOrCreate()

path = 'E:\skola\DP\logovani_ublox\data_bez_cisel\proc_checked.csv'

coords = sc.read.csv(path, sep=',', header=True)
```

Data je třeba přetypovat. K tomu je využito příkazu *.withColumn()*. Prvním argumentem je název sloupce, do kterého budeme změny ukládat, a druhým argumentem je hodnota či příkaz, který provádíme nad zadanými sloupci.

Jelikož není potřeba pro tuto úlohu všech sloupců, jsou vybrány jen ty, které vstupují do výpočtu. V tomto případě jsou to sloupce obsahující čas měření, zeměpisnou šířku, zeměpisnou délku a výšku.

```
coords = coords \
    .withColumn("start_time", col("start_time").cast(TimestampType())) \
    .withColumn("lat_mean", col("lat_mean").cast(DoubleType())) \
    .withColumn("lat_std", col("lat_std").cast(DoubleType())) \
    .withColumn("lon_mean", col("lon_mean").cast(DoubleType())) \
    .withColumn("lon_std", col("lon_std").cast(DoubleType())) \
    .withColumn("alt_mean", col("alt_mean").cast(DoubleType())) \
    .withColumn("alt_std", col("alt_std").cast(DoubleType()))

coords_red = coords.select("start_time", "lat_mean", "lon_mean",
"alt_mean")
```

Pro lepší orientaci ve výsledcích a zpřehlednění kódu je provedeno přejmenování sloupců. K tomu je užito příkazu pro vytvoření nového DataFramu *.toDF().*

```
newColumns = ["start_time", "lat", "lon", "alt"]
coords_ren = coords_red.toDF(*newColumns)
```

Jelikož je v této úloze požadavek na spočítání odchylek od průměru, je třeba již zmíněný průměr nejprve spočítat. Zápis pro uložení této hodnoty je v PySparku složitější záležitostí. Důvodem je skutečnost, že veškeré agregační funkce ve Sparku vracejí zpět data ve formě DataFramu. Problémem DataFramu ve Sparku je to, že je nelze jednoduše indexovat. Proto, pokud je požadavek na práci s jednou hodnotou a její uložení do samostatné proměnné typu např. double, je třeba provést sadu příkazů nad vstupním DataFramem.

V PySparku, pokud chceme použít agregační funkci, je třeba nejprve DataFrame seskupit příkazem *.groupby().* Jeho argumentem je obecně sloupec, podle kterého seskupení provádíme. Nad takto

seskupenými daty již provádíme onu agregaci, v tomto případě průměr. Průměr lze spočítat pomocí příkazu *.mean()* či *.avg()*.

Příkaz *.take()* vrací daný počet řádků jako formát *list*. Jelikož je zde požadavek na vložení do proměnné jako konstanty, je třeba dodat za příkaz [0][0].

```
meanLat = coords_ren.groupby().mean("lat").take(1)[0][0]
meanLon = coords_ren.groupby().mean("lon").take(1)[0][0]
```

Pro zrychlení výpočtu jsou vloženy průměrné hodnoty zeměpisné šířky a zeměpisné délky do samostatných sloupců v DataFramu. K tomu je použit příkaz *.withColumn().* Prvním argumentem je název nového sloupce a druhým je předem vypočtená konstanta. Konstanty jsou vkládány do DataFramu pomocí funkce *lit()*.

Samotný výpočet je opět proveden pomocí příkazu *.withColumn()*. V tomto případě jsou na sebe navázány dva tyto příkazy, kde jeden je pouze mezivýpočtem pro spočítání skutečné vzdálenosti.

```
coords_ren = coords_ren\
    .withColumn("latMean", lit(meanLat))\
    .withColumn("lonMean", lit(meanLon))
# Spherical distance calcualtion based on latitude and longitude with
Apache Spark (inspiration: pavlov99, harpaj)
coords_dist = coords_ren.withColumn("a", (
        pow(sin(radians(col("lat") - col("latMean")) / 2), 2) +
        cos(radians(col("latMean"))) * cos(radians(col("lat"))) *
        pow(sin(radians(col("lon") - col("lonMean")) / 2), 2)
    )).withColumn("distance_deviation[m]", atan2(sqrt(col("a")), sqrt(-
col("a") + 1)) * 12742000)
```

Pro vypsání hodnot seřazených podle vzdálenosti od průměru je využito příkazu *.orderBy()*, jehož argumentem je sloupec obsahující odchylky seřazené sestupně podle hodnoty.

```
coords_dist.select("start_time",
"distance_deviation[m]").orderBy(col("distance_deviation[m]").desc()).show(
)
```

Výstup úlohy vypsaný v terminálu:

```
+-------------------+--------------------+
|       start_time|distance_deviation[m]|
+-------------------+--------------------+
|2021-12-20 11:07:48|   4.782321442519063|
|2021-12-10 13:33:46|  3.5482996426564237|
|2022-01-13 05:55:27|    3.54440183426356|
|2021-11-26 02:52:31|  3.1171939786981255|
|2021-12-07 11:50:58|  2.3585664679937497|
.....
```

4.3.4. Histogram

Pro ukázku tvorby grafických výstupů bylo vybráno zobrazení rozptylu odchylek pomocí sloupcového grafu. Protože při nastavení zobrazení histogramu přes celou datovou sadu docházelo k zamrznutí stroje, byla datová sada omezena na odchylky menší než 0.01m.

```
coords_graph = coords_dist.where(col("distance_deviation[m]") < 0.01)
```

Samotný PySpark má prostředky k vypsání histogramu, nelze jej však poté graficky zobrazit. Nejprve tedy vypíšeme výsledky do terminálu.

Pro výpočet histogramu se používá funkce *.histogram()*, jejímž argumentem je počet intervalů, do kterých se data rozdělují. Nejprve je však třeba vybrat data, která jsou základem pro jeho tvorbu. Použijeme pro to příkaz *.select()* a data takto vybraná následně převedeme na RDD. Převod na RDD se provádí proto, že *histogram* není definovaný pro DataFrame.

```
dist_histogram =
coords_graph.select(round('distance_deviation[m]',4)).rdd.flatMap(lambda x:
x).histogram(20)
print(dist_histogram)
```

Výstup příkazu *print.(„histogram“)*.

```
([1.4301871757774477e-05, 0.0005134975666931073, 0.00101269326162844,
0.001511888956563773, 0.0020110846514991058, 0.002510280346434486,
0.0030094760413697714, 0.003508671736305104, 0.004007867431240475,
0.00450706312617577, 0.005006258821111103, 0.005505454516046436, 0.006004650210981769,
0.006503845905917101, 0.007003041600852434, 0.007502237295787768, 0.0080014329907231,
0.008500628685658432, 0.008999824380593765, 0.009499020075529099,
0.009998215770464431],

[52, 137, 252, 331, 391, 486, 501, 584, 588, 611, 610, 630, 638, 600, 574, 542, 490, 446, 422, 398])
```

4.3.5. Grafické zobrazení histogramu

Vzhledem k tomu, že podle dostupných zdrojů a dokumentací PySpark současně nemá impelemntovanou komponentu GraphX, je třeba pro zobrazení grafu využít jiných knihoven, které Python nabízí.

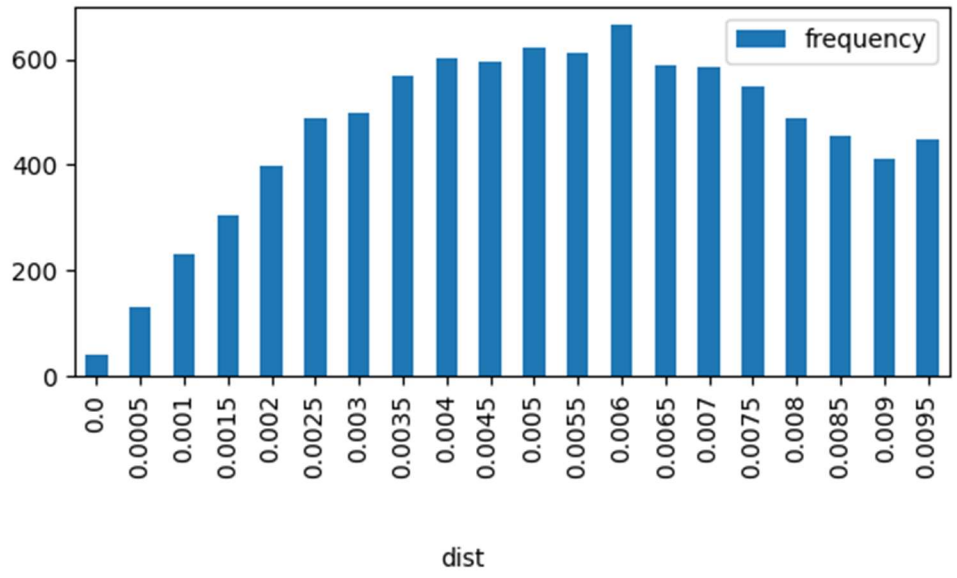
Jde o již zmiňovanou knihovnu Matplotlib, která umožňuje tvorbu grafů. Jelikož Matplotlib je jedním ze základů Pandas, použijeme Pandas DataFrame k vykreslení a uložení dat. Funkce *zip()* se používá ke spojení dvou či více souborů dat fomrátu *list.* Z tohoto spojení se poté vytvoří pomocí příkazu *DataFrame* stejnojmenný objekt. DataFrame má dva sloupce, jedním je hodnota vzdáleností a druhým četnost výskytu.

Thákurova 7, 166 29 Praha 6

Dále je třeba určit hlavní osu a typ zobrazení. V tomto případě je hlavní osou osa s hodnotami vzdáleností a typem grafu je graf sloupcový. Pro zobrazení grafu se využije příkaz *.show().*

```
pd.DataFrame(
    list(zip(*dist_histogram)),
    columns=['dist', 'frequency']
).set_index(
    'dist'
).plot(kind='bar')
plt.show()
```

Výsledný sloupcový graf:



Graf 2 Sloupcový graf odchylek vytvořený pomocí Pyspark

U dvojrozměrné a trojrozměrné transformace byl zaznamenán minimální rozdíl výsledných souřadnic. Po přezkoumání byla odhalena odchylka konstantní, která je nejspíše způsobena zapojením výšky do transformace.

Tabulka 1 Odchylky transformací

	3D transformace		2D transformace		Rozdíly	
start_time	Y	X	Y	X	ΔY	ΔX
2021-11-05 11:01:16	-744928,6489	-1040875,898	-744928,6461	-1040875,895	-0,002820751	-0,00290842
2021-11-05 11:01:16	-744928,6568	-1040875,916	-744928,654	-1040875,913	-0,002820904	-0,00290858
2021-11-05 11:01:16	-744928,658	-1040875,898	-744928,6552	-1040875,895	-0,002820759	-0,00290843
2021-11-05 11:01:16	-744928,6479	-1040875,913	-744928,6451	-1040875,91	-0,002820797	-0,00290848
2021-11-05 11:01:16	-744928,6557	-1040875,915	-744928,6529	-1040875,912	-0,002820761	-0,00290843

5. Závěr

Cílem této práce bylo zjistit, jestli je Apache Spark vhodným zpracovatelským prostředím pro data v oblasti geodézie a kartografie. Práce seznamuje v první polovině čtenáře s pojmem Big data a nástroji vhodnými k jejich zpracování. Přináší pohled na rozdělení, fungování a zdroje Big dat i na architekturu, využití a obecně fungování nástrojů Apache Hadoop, jeho nástupce Apache Spark a jeho použití společně s jazykem Python. Ve druhé polovině se práce zaměřuje na praktické ukázky využití v oblasti geodézie a kartografie.

Ukázky obsahují aplikaci základních geodetických úloh v prostředí PySpark. Úlohy se zaměřují na zpracování velkého množství měření. Tato měření mohou přicházet v reálném čase. Vzhledem k absenci stroje či uložiště, na které by se dalo v připojit a ze kterého by proudila současně měřená data, jsou data nahrávána z lokálního disku.

5.1. Porovnání s klasickými nástroji na zpracování dat

Apache Spark je analytický engine pro zpracování velkého množství dat v krátkém čase. Toto zvládá velice dobře a je tomu uzpůsoben. V oblasti geodézie a kartografie narážíme v dnešní době na velká data velice často, a právě pro takováto data je využití Sparku jako zpracovávatelského nástroje ideální. Například výpočet posunu bodů, nalezení odlehlých měření, počítání souřadnic bodů v polygonovém pořadu atd., to vše jsou úlohy, které lze zpracovávat v reálném čase a vyhodnocovat téměř v okamžiku jejich naměření. Pokud tedy máme velké množství dat, použití Sparku se časově vyplatí oproti klasickým nástrojům, do kterých by bylo potřeba data importovat z lokálního disku jeden soubor po druhém, či soubory manuálně spojovat ještě před importem. Do zpracovávatelského prostředí Apache Spark lze data nahrát přímo ze vzdáleného uložiště, bez nutnosti meziukládání na lokální disk. Další výhodou je již zmíněná možnost připojení real-time na stroj či úložiště a přebírání dat v okamžiku, kdy jsou nasnímána nebo uložena. Zde nastává teoretická situace, kdy budou měřeny například souřadnice bodů na hrázi. Data budou do výpočtu vstupovat v reálném čase každou sekundu a rovnou budou vypočítávány jejich odchylky od průměru či jejich známých souřadnic. Spark jakožto analytický engine podporuje i možnost SQL, což je velmi mocný nástroj pro zpracování a vyhodnocování dat, který klasické nástroje nemají. Při užití Sparku pro zpracování dat je samozřejmostí, že si uživatel může, a v současné době musí, napsat všechny funkce sám. Není tedy limitován schopnostmi programu, za který si zaplatil, a v případě potřeby nemusí vyhledávat jinou aplikaci, která jeho požadavek dokáže splnit. Finální velkou výhodou je samozřejmě to, že uživatelem napsané aplikace může využít zdarma.

Spark však je určen pro analýzu dat obecně a nedisponuje proto předdefinovanými funkcemi či přímo knihovnami pro zpracování složitějších geodetických úloh. Uživatel je tedy odkázán sám na sebe a své schopnosti, popřípadě na informace a rady napsané na internetu. V současné době si zkrátka musí své funkce napsat sám. Python obsahuje vskutku obrovské množství knihoven, které lze volně použít a kombinovat se Sparkem (PySparkem). Tyto knihovny lze velmi efektivně využít při zpracování prostorových dat, avšak povětšinou nejsou primárně k tomu určeny. Výjimkou je například knihovna PyPROJ, která je knihovnou kartografických projekcí a transformací souřadnic. Spark používá k ukládání dat a práci s nimi již zmíněnou strukturu DataFrame, která organizuje data do dvojrozměrné tabulky. Tato tabulka však nepodporuje například indexování, a proto přístup

k jednotlivým datům může být problematický. Rovněž jakýkoliv výběr z DataFramu je uložen v základu znova jako DataFrame, což působí problémy například při požadavku vybrat jednu hodnotu jako samostatnou proměnnou. Další problémy nastávají při samotném nahrávaní dat do DataFramu, neboť bez uvedení schématu dat či hlavičky jsou data uložena vždy jako textový řetězec. Při práci s numerickými daty je tedy vždy potřeba vytvořit schéma a definovat typy dat, které vstoupí do struktury, anebo přetypovat data po jejich nahrání. Rovněž výsledné zobrazení či výpis výsledků jsou možné pouze ve formě DataFramu, nebo jako export do jiných formátů například csv či txt. Spark ve spojení s Pythonem současně nepodporuje svou součást GrapX a není tedy zatím možné vykreslovat s jeho pomocí grafy, ani jinak graficky zobrazovat data.

5.2. Zhodnocení využití Apache Spark v geodézii a kartografii

Využití Apache Spark jako nástroje pro zpracování je samozřejmě vhodné a výhodné, pokud máme k dispozici velký soubor dat. Pokud bychom měli užívat Apache Spark kvůli stovce záznamů, bylo by to v podstatě zbytečné a použití klasických nástrojů se vyplatí daleko více. Pokud budeme získávat data v reálném čase, či budeme data přebírat ze vzdáleného serveru, je nejen časově výhodné využít napojení přímo na stroj či úložiště bez nutnosti data znova stahovat a ukládat. Při snaze o analýzu informací, vztažených k prostorovým informacím, je Spark rovněž velmi efektivním nástrojem. Umožní nám provádět analýzu jak na prostorových informacích, tak na datech obecného charakteru najednou, což je velkou výhodou.

Obecně je tedy využití Apache Spark velkým přínosem i přes všechny jeho nedostatky. K efektivnímu používání Apache Spark ve spojení s jazykem Python v oblasti geodézie a kartografie chybí pouze pár kroků, a to vytvoření knihovny, obsahující geodetické úlohy či možnost graficky vykreslovat zpracovávaná data.

Využité knihovny pro Python

Seznam níže obsahuje knihovny použité v této práci a odkaz k jejich dokumentaci. Počet použitých knihoven je ve skutečnosti mnohem větší, avšak jelikož jsou zbylé knihovny importovány spolu s těmi uvedenými níže jako jejich součásti, nejsou v tomto seznamu uvedeny.

PySpark	https://spark.apache.org/docs/latest/api/python/
PyPROJ	https://pyproj4.github.io/pyproj/stable/
Sshtunnel	https://pypi.org/project/sshtunnel/
Py4J	https://www.py4j.org/
Pandas	https://pandas.pydata.org/
NumPy	https://numpy.org/
Matplotlib	https://matplotlib.org/
PyMySQL	https://pypi.org/project/PyMySQL/

Seznam obrázků

Bibliografie

1. ***Statista.*** **[Online] [Citace: 10. 5 2022.] https://www.statista.com/chart/17723/the-data-created-last-year-is-equal-to/.**

2. Mani Mandhukar, Pooja. *Eartg Science (Big) Data Analytics.* 2018.

3. Doukas, Ioannis D. RE-DISCOVERING "BIG DATA" AND “DATA SCIENCE” IN GEODESY AND GEOMATICS. [Online] 15. Květen 2019. https://www.fig.net/resources/proceedings/2019/04_JISDM2019/42.pdf.

4. Mike Smith, Paolo Paron, James Griffiths. *Geomorphological Mapping - Methods and Applications.* October 22, 2011. 9780444534460.

5. Apache spark. *Apache spark.* [Online] spark.apache.org.

6. *python.org.* [Online] [Citace: 25. 4 2022.] https://www.python.org/doc/essays/blurb/.

7. *Py4J.* [Online] [Citace: 25. 4 2022.] https://www.py4j.org/.

8. Apache Hadoop Core. *Hadoop.* [Online] hadoop.apache.org.

9. aniruddha. *analyticsvidhya.* [Online] 23. Říjen 2020. https://www.analyticsvidhya.com/blog/2020/10/introduction-hadoop-ecosystem/.

10. Apache Spark with Scala - Hands On with Big Data! *udemy.com.* [Online] https://www.udemy.com/course/apache-spark-with-scala-hands-on-with-big-data/.

11. Schmidt, Jeroen. *towardsdatascience.* [Online] [Citace: 15. 4 2022.] https://towardsdatascience.com/apache-spark-101-3f961c89b8c5.

12. aws.amazon.com/big-data/. *aws.amazon.* [Online] aws.amazon.com/big-data/.

13. PyMySQL. *PyMySQL.* [Online] [Citace: 2. 5 2022.] https://pymysql.readthedocs.io/en/latest/index.html.

14. pyproj. [Online] [Citace: 3. 5 22.] https://pyproj4.github.io/pyproj/stable/index.html.

15. *SparkByExamples.* [Online] [Citace: 3. 5 2022.] https://sparkbyexamples.com/.

16. *Matplotlib.* [Online] [Citace: 5. 5 2022.] https://matplotlib.org/.