Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra počítačových systémů

Diplomová práce

Rozšíření projektu Spark o podporu jazyka Ruby

Bc. Ondřej Moravčík

Vedoucí práce: Ing. Tomáš Bartoň

30. dubna 2015

Poděkování

Chtěl bych poděkovat mému vedoucímu práce Tomáši Bartoňovi, který navrhl toto velice zajimavé téma diplomové práce a hlavně za to, že mi ukázal a představil projekt Apache Spark.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 30. dubna 2015 .....

Odkaz na tuto práci

Moravčík, Ondřej. *Rozšíření projektu Spark o podporu jazyka Ruby.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Tato práce se zabývá rozšířením projektu Apache Spark o podporu jazyka Ruby. Součásti práce je také analýza různých metod serializace včetně porovnání výkonu s ostatními jazyky.

Klíčová slova Ruby, Spark, Distribuované systémy, Distribuované výpočty

Abstract

This magister thesis deals with the extension of Apache Spark to supporting Ruby language. Thw work includes analysis of various serialization methods and comparison with other languages.

Keywords Ruby, Spark, Distributed systems, Distributed calculations

Obsah

Seznam obrázků

Úvod

V současné době jsou pojmy jako *Clusterovaná řešení*, *Distribuované výpočty*, *Data-mining* nebo *Machine learning* velmi skloňovaným tématem. Dříve byly tyto pojmy důležité hlavně v IT oblasti, ale dnes si všechny firmy uvědomují, že data již nejsou jen nevýznamné byty, které zabírají místo. Naopak to jsou velmi cenná aktiva, které je nutné chránit stejně jako hmotný majetek. Surová data mají ale většinou nulový význam, pokud nejsou zpracována do využitelné podoby. Dat je ale obvykle velmi mnoho a tudíž je nutné, aby bylo zpracování distribuované na více počítačů.

> *Data is the new oil* [1]

Na trhu lze najít spousty produktů podporujících specializované výpočty či distribuci dat. Obvykle jsou tyto nástroje odděleny a každý se specializuje na danou věc.

Výpočty

- RapidMiner
- IPython
- Spark
- Microsoft Excel

Distribuce

- Hadoop
- YARN
- Mesos

Problém je najít vhodný nástroj, který umožní zpracovávat data tak, jak potřebujeme, bude dostupný pro náš jazyk a zároveň umožní paralelizaci výpočtů.

Jednoznačně nejpopulárnější jazyk v těchto oblastech je Java. Ten nabízí asi nejlepší poměr snadná implementace/správa vůči rychlosti. Naštěstí se i v této oblasti začínají vyvíjet nástroje podporující jiné jazyky jako je Python nebo R.

Ruby bohužel zůstává v této oblasti pozadu. Touto magisterskou práci se to pokusím změnit přidáním podpory jazyka Ruby do projektu Apache Spark. Výsledek magisterské práce bude volně dostupná knihovna s názvem RubySpark.

Kapitola 1

Apache Spark

V roce 2002 publikoval Google techniku zvanou MapReduce. Ta měla za cíl poskytnout nástroje pro distribuované výpočty. Tento postup lze zjednodušeně popsat a rozdělit na 2 fáze:

Map: mapování dat na uzly v síti a výpočty na nich

Reduce: redukování výsledku zpět

O 4 roky později vznikla první verze systému Hadoop, který tuto techniku implementoval. Data byly uloženy a distribuovány na souborovém systému (HDFS), který byl součásti Hadoopu. Později se ale ukázalo, že definování výpočtu nad tímto systémem je velice složité. Navíc byly data na disku a tak rozsáhlejší výpočty byly velice pomalé. Z těchto důvodů začaly vznikat specializované nástroje soustředící se pouze na jeden druh výpočtu (například Pregel, Storm nebo Giraph).

V roce 2009 vznikl projekt Apache Spark, který měl problémy odstranit pomocí:

- data jsou uložena v paměti nebo na disku
- data v paměti mohou být uložena jako Javovské struktury
- poskytování knihoven pro různé druhy výpočtů
- data jsou rozděleny na části a ty pak distribuované

Dnes je Spark rychlý, všeobecně použitelný výpočetní systém napsaný v jazyce Scala. Zobecňuje použití MapReduce a umožňuje spouštět výpočty jak lokálně, tak na několika typech clusterů, které jsou popsány v kapitole 2.

Projekt se v tuto chvíli dělí na Obecné výpočty, Mllib, Streaming, SQL a GraphX a podporuje použití Javy, Scaly a Pythonu. Od verze 1.4 navíc přibude jazyk R, který už nebude jako samostatný projekt, ale stane se součástí jádra.

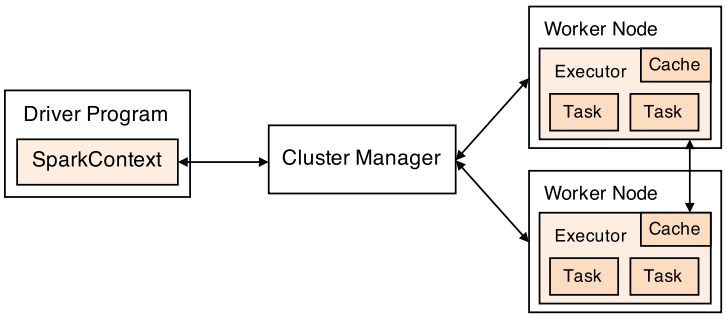
Hlavní vývoj probíhá ve Scale a tak společně s Javou podporují veškerou funkcionalitu. Python je brán jako vedlejší a tudíž nepodporuje vše. Předchozí jazyky jsou navíc kompilované a Python intepretovací s C interpretem. Tudíž nelze jednoduše převádět funkce a datové struktury tam a zpět. Nastalý problém se řeší dvěma způsoby.

1. přepsáním stávajících metod do nového jazyka
2. serializací dat do struktury Javy

Druhý způsob se převážně využívá v knihovně Mllib a proto nastalé problémy a výhody jsou popsány v kapitole 1.3. Na většinu funkcí je použit první způsob.

Spark je koncipován tak, aby uživateli nabídl rozhraní, které se nebude odlišovat od toho, na který je zvyklý v daném jazyku. To je docíleno stejným pojmenováním a podobnou či stejnou funkcionalitou metod. Jediný rozdíl je, že metody nejsou prováděny na běžné kolekci dat, ale na paralelizované.

Tyto data se označují jako RDD (*„resilient distributed dataset”*). Je to kolekce, rozdělená napříč výpočetními uzly, které počítají paralelně (nezávisle na sobě). Vznikne buď načtením souborů, nebo pomocí přímého vložení existující kolekce v řídícím programu. Pro efektivnější práci lze využít uložení v paměti nebo sdílených proměnných.



Obrázek 1.1: Clustering

Výpočty mohou být spuštěny na 3 typech clusterů.

- Lokální
- Mesos
- YARN

1.1 Instalace

Požadavky

- Java 6+
- Python 2.6+
- Scala 2.10, 2.11

Podporovány jsou operační systémy UNIX i Windows. Instalaci lze provést 3 různými způsoby.

1.1.1 Pomocí balíčku

Na adrese spark.apache.org lze stáhnout připravené balíčky s konkrétní verzí Spark a Hadoop. Tyto balíčky jsou připravené pro Scalu 2.10. Pro jinou verzi je nutné zvolit jiný postup.

1.1.2 Linkování pro Java projekt

Spark lze také snadno přidat do Java projektu, který pro sestavení využívá Maven, SBT, Ivy, ... Připravené balíčky jsou uložené v oficiálním Maven repozitáři. Nastavení závislosti:

```
groupId: org.apache.spark
artifactId: spark-core_2.10
version: 1.3.0
```

1.1.3 Kompilace zdrojových kódů

Apache Spark je šířen pod open-source knihovnou a tudíž lze použít poslední vývojovou verzi, která je dostupná na veřejném GIT repozitáři. Pro projekt jsou dostupné sestavovací skripty pro Maven a SBT.

Sestavení celého projektu včetně všech knihoven lze provést pomocí:

```
sbt/sbt assembly
```

1.2 Použití

Všechny příklady, které budu uvádět, jsou v programovacím jazyce Python, protože je nejblíže k Ruby.

Před spuštěním samotného Sparku je nejdříve nutné vytvořit konfiguraci. Ta se po spuštění (vytvoření Contextu) již nedá změnit. Konfiguraci lze nastavit předem v systémových proměnných nebo pomocí `SparkConf`.

```
conf = SparkConf().setAppName(appName).setMaster(master)
```

Možností nastavení je opravdu mnoho a všechny vypisovat nebudu. Ty nejpodstatnější jsou v následující tabulce.

Klíč	**Výchozí**	**Popis**
`spark.app.name`	*nic*	Jméno aplikace, použité v WebUI a v logu.
`spark.master`	*nic*	Cluster manager, ke kterému se Spark připojí.
`spark.executor.memory`	512m	Maximální velikost paměti pro výpočetní proces.
`spark.driver.memory`	512m	Maximální velikost paměti pro řídící proces.
`spark.serializer`	JavaSerializer	Třída, která je použitá pro serializaci.
`spark.local.dir`	/tmp	Složka, kde budou odkládány dočasné data.
`spark.python.worker.memory`	512m	Maximální velikost paměti pro Python výpočetní proces.
`spark.default.parallelism`		Výchozí počet části na, které bude RDD rozdělen. Výchozí hodnota závisí na zvoleném clusteru nebo na počtu CPU.

Jakmile je konfigurace hotova, lze vytvořit `SparkContext`. Je to hlavní přístupový bod k celé funkcionalitě Sparku. Reprezentuje připojení do zvoleného clusteru a lze v něm vytvářet objekty jako RDD, Broadcast a Accumulator. Pro jednu instanci JVM lze vytvořit a mít aktivní pouze jeden context.

```
sc = SparkContext(conf=conf)
```

1.2.1 RDD

Reprezentace paralelizovaných dat, nad nimiž lze provádět výpočty.

Tato třída může vzniknout několika způsoby, ale nejpoužívanější je z lokální kolekce nebo ze souboru. U každého způsobu lze navíc zvolit, na kolik části budou data rozdělena pomocí parametru `PARTITION_NUM`. V takovém případě bude vytvořeno tolik výpočetních procesů, aby každý dostal právě jeden.

Z lokální kolekce

```
data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data, PARTITION_NUM)
```

Ze souboru

```
rdd = sc.textFile("data.txt", PARTITION_NUM)
```

U druhého způsobu představuje položka v kolekci řádek v souboru **data.txt**. Soubor může být vložen jak z lokálního souborového systému, tak z extérního (například HDFS - *Hadoop File System*).

Nad proměnnou `rdd` lze nyní volat stejné operace jako u tradičního pole. Rozdíl je, že vložené funkce jsou řetězeny za sebou a následně jsou i s daty poslány k výpočetním procesům. Každá nově přidaná funkce není poslána automaticky.

Pro spuštění výpočtu je nutné zavolat funkci `.collect()`. Ta je navíc dostupná pro každý článek v řetězci příkazů. Na prvním článku dostaneme zpět naše původní data. Některé operace (například: `.reduce()`) volají `.collect()` samy.

Struktury dat

V celém Sparku lze data rozdělit do mnoha druhů. Například specializované struktury pro SQL či Streaming. Teď nám ale postačí základní rozdělení na obyčejné a typ key-value.

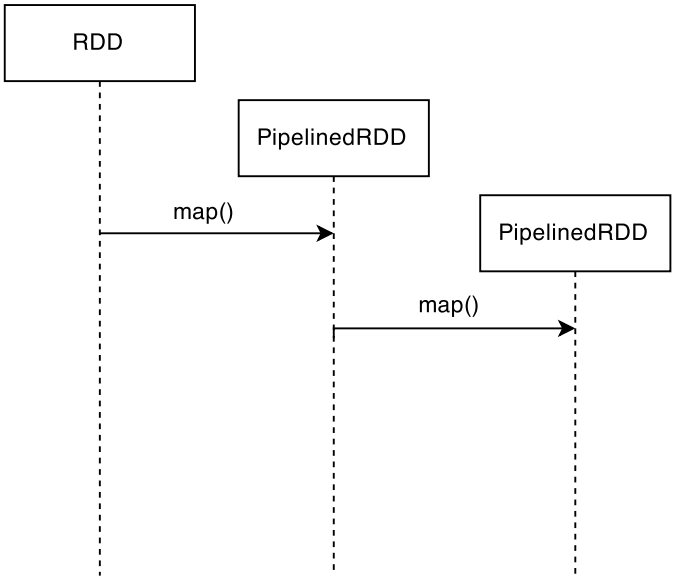
Typ key-value nemá žádnou speciální definici. Je to vlastně jen pole o 2 prvcích, kde první reprezentuje klíč a druhý hodnotu. Lze ji jednoduše deklarovat pomocí

```
keyValue = [(key1, value1), (key2, value2)]
```

Zřetězení operací

Všechny metody mají následující formát. První článek (`RDD`) reprezentuje původní nezměněné data.

```
RDD.OPERACE_1.OPERACE_2.OPERACE_3.COLLECT
```



Obrázek 1.2: Pipelinování RDD

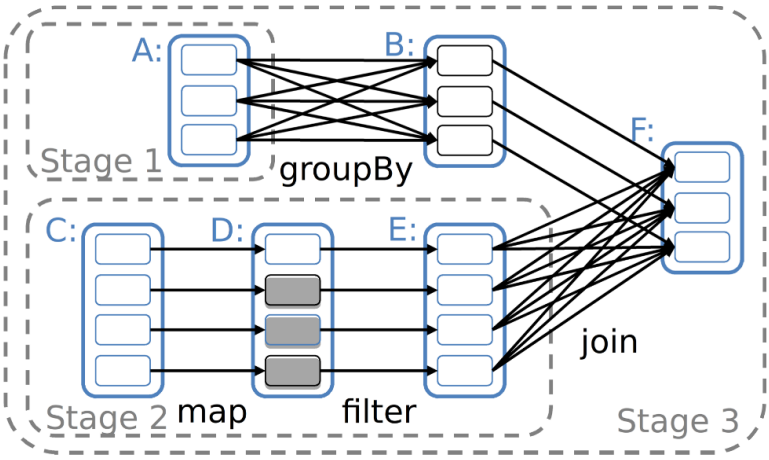
U takto zřetězených příkazů je nutné dát pozor, na jaký článek voláme funkci `.collect()`. Ta do výpočtu pošle nejen příkaz, ze kterého článek vznikl, ale také všechny předky včetně původního.

Teoreticky tento řetězec může mít neomezenou velikost, nicméně se doporučuje nedělat to. Velká část příkazu prochází každou položku pole a tak více příkazů znamená více průchodů stejným polem. V takovém případě je lepší použít funkce, které na vstupu dostanou rovnou celou kolekci místo jednotlivých položek.

1.2.1.1 Výpočty

Podkapitola popisuje situaci potom co na naší kolekci dat zavoláme `.collect()`. Například máme dvě sady, které vzniknout pomocí.

```
rddAB  = rdd.groupBy(...).collect()
rddCDE = rdd.map(...).filter(...).collect()
rddF   = rddAB.join(rddCDE)
```



Obrázek 1.3: Schéma jednoduchého výpočtu

Stage V tomto případě představuje výpočetní úkol a políčka v něm části kolekce.

A-B: **A** obsahují různé hodnoty, které chceme seskupit podle definované funkce. Například pokud máme pole slov, tak je seskupíme podle jejich délky.

```
[a, aaa, bbb] -> [(1, a), (3, aaa, bbb)]
```

C-D: Na každý element **C** je aplikována funkce. Například vytvoření key-value struktury, kde klíč je délka slova.

```
[aa, bbb, ccc] -> [(2, aa), (3, bbb), (3, ccc)]
```

D-E: Z kolekce **D** jsou vybrány pouze hodnoty odpovídají definovanému pravidlu. Například délka musí být větší než 2.

```
[(2, aa), (3, bbb), (3, ccc)] -> [(3, bbb), (3, ccc)]
```

BE-F: Dvě kolekce jsou seskupeny podle klíče. Tomuto procesu se říká shuffle.

```
[(1, a), (3, aaa, bbb)], [(3, bbb), (3, ccc)] ->
[(1, a), (3, aaa, bbb, bbb, ccc)]
```

Musíme brát na vědomí, že jednotlivé procesy spolu nemohou komunikovat a pracují zcela nezávisle nad různými daty. S tím je nutné počítat hned na začátku a podle toho přizpůsobit výpočetní metodu a počet částí, na které budou data rozdělena.

Shuffle
Jak již název napovídá, tato operace se zabývá mícháním dat. Obecně je metoda dostupná nad polem dat a zařídí, aby byly všechny položky náhodně zpřeházené.

Ve Sparku ovšem míchání není náhodné, nýbrž podle přesně definovaných pravidel a postupů. Slouží k tomu, aby se všechny stejné hodnoty dostaly do jednoho uzlu. Například budu-li mít kolekci key-value dat rozdělené mezi dvěma výpočetními procesy a bude žádoucí je seskupit.

```
Proces 1: [(1, a), (2, b), (3, c)]
Proces 2: [(1, d), (4, e)]
```

Naivní řešení by bylo projít všechny položky a ty nějak rozdělit. Musíme mít na paměti, že výpočetní procesy posílají serializované výsledky. Tedy jako pole bytu. To by ale znamenalo deserializovat hodnoty a ty pak rozdělit. U složitých či objemných dat by tato operace představovala příliš velkou zátěž a proto se před odesláním výsledku připojí ke každé položce hash-klíč.

Hash-klíč je speciální hodnota velikosti 8B, která charakterizuje hodnotu. Sparku v takovém případě stačí načíst pouze klíče a pokud se nějaké shodují tak je pošle na stejný uzel.

Aby bylo toho řešení efektivní, musí být splněny následující podmínky.

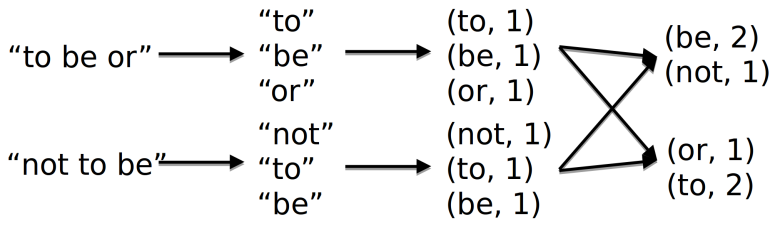
- každý výpočetní proces musí vytvářet hash-klíče podle stejného schématu
- výpočet klíče musí být co nejrychlejší i za cenu kolizí

1.2.1.2 Příklady

Jakmile máme kolekci dat, lze volat příslušné operace. Jednoduchý příklad počítání slov vypadá následovně.

Počet slov v souboru

```
file = spark.textFile("hdfs://...")

file.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a+b)
```



Obrázek 1.4: Fáze počítání počtu slov

1. Načtení dat ze souboru.
2. Rozdělení každého řádku na jednotlivá slova.
3. Každé `slovo` se nahradí za `(slovo, 1)`
4. Všechny struktury, se stejným slovem se seskupí na jeden uzel, který je nahradí za `(slovo, SOUČET VŠECH 1)`

Samozřejmě tento příklad je možno provést efektivněji, ale dává názorný příklad jak pracovat se Sparkem.

Transformace Obecné metody pro transformaci dat. Ty se pouze řetězí a nespouští výpočty.

Metoda	Popis
`map(func)`	Na každou jednotlivou položkou kolekce je volána func.
`flatMap(func)`	Podobné jako `.map()`. Všechny vnořené pole se navíc sloučí do jediného.
`filter(func)`	Z kolekce jsou vybrány pouze položky pro, které vrací func pravdu.
`mapPartitions(func)`	Podobné `.map()`. Func jako parametr nedostává jednu složku, ale celé pole určené pro daný výpočetní proces.
`mapPartitionsWithIndex (func)`	Podobné `.mapPartitions()`. Navíc přibyl druhý parametr a to index dané kolekce.
`sample(repl, fr, s)`	Vytvoření vzorku z kolekce podle parametrů *fraction* a *seed*.
`union(other)`	Vrátí novou kolekci obsahující data se současné a *other* kolekce.
`intersection(other)`	Vrátí průnik dvou množin.
`distinct(num))`	Vrátí pouze unikátní (jedinečné) položky.
`groupByKey(num)`	Seskupí všechny data podle klíče (první položka v poli.)
`reduceByKey (func, num)`	Redukování dat podle klíče a vložené funkce.
`aggregateByKey(zero) (seq, comb, num)`	Redukování podle klíče. **zero** inicializační hodnota pro seskupení hodnot **seq** funkce pro redukování výsledku ve výpočetním procesu **comb** funkce pro redukování všechn výsledku, ze všech procesů **num** kolik procesů bude danou ulohu provádět
`sortByKey(asc, num)`	Řezení dat podle klíče.
`cogroup(other, num)`	Spojení kolekcí podle jejich klíču. Datasety ve tvaru `(K, V)` a `(K, W)` se spojí do `(K, (V, W))`.

`cartesian(other)`	Kartézký součin dvou kolekcí.
`pipe(command, env)`	Výpočetní proces spustí na pozadí systémový příkaz *command* do kterého bude vkládat svojí kolekci.
`coalesce(num)`	Zmenšení počtu rozdělení původní kolekce.
`repartition(num)`	Zmenšení nebo zvětšení počtu části. Počet elemntů je rovnoměrně rozdělen mezí všechny procesy.

Akce Obecné akce podporované Sparkem. Po jejich zavolání dojde výpočtu a vrácení výsledku.

Metoda	**Popis**
`reduce(func)`	Agregování kolekce podle dané funkce.
`collect()`	Vrátí všechny položky zpět do řídícího programu.
`count()`	Počet položek v kolekci.
`first()`	První položka kolekce.
`take(n)`	Vrátí n položek kolekce.
`takeSample (repl, num, s)`	Vrátí vzorek kolekce o velikosti *num*.
`takeOrdered(n, order)`	Vrátí n seřazených položek.
`saveAsTextFile(path)`	Uloží celou kolekci do zadaného souboru.
`countByKey()`	Počet hodnot, ke každému klíči.
`foreach(func)`	Nad každou položkou je voláná funkce *func*. Na rozdíl od `.map()` nevrací výpočetní proces výsledek. Metoda může sloužit například k aktualizování akumulátoru nebo k interakci se systémem.

1.2.2 Cache

Velice důležitou součásti Sparku je možnost trvalého uložení výsledku do zvolené paměti. V takovém případě si bude Spark držet výsledek a nebude muset znova spouštět výpočty. Typy:

`MEMORY_ONLY` Uložení deserializovaných objektů v JVM. Pokud se některé objekty nevejdou do paměti, budou zahozeny a vypočítány znova jakmile bude třeba (výchozí hodnota).

`MEMORY_AND_DISK` Stejný případ jako `MEMORY_ONLY`. Nicméně, pokud se data nevejdou do paměti, jsou uložena na disku a přečtena, když je třeba.

`MEMORY_ONLY_SER` Uložení serializovaných objektů (pole bytů). Tato metoda je výhodnější ve využití paměti, nicméně náročnější na procesorový čas.

`MEMORY_AND_DISK_SER` Stejné jako `MEMORY_ONLY_SER`. Místo zahození se data ukládají na disk.

`DISK_ONLY` Uložení dat pouze na disk.

`MEMORY_ONLY_2, MEMORY_AND_DISK_2, ...` Podobné jako předchozí příklady. Data jsou ale rozdistribuovaná na 2 výpočetní uzly.

`OFF_HEAP` Uložení dat v Tachyon formátu (experimetální).

V ostatních jazycích kromě Scaly a Javy jsou data vždy serializována jako pole bytů.

V některých případech je tato funkce volána automaticky. Například při operacích redukce.

1.2.3 Sdílené proměnné

Někdy existuje proměnná, která je využívána všemi výpočetními úkoly. V normálním případě by musela být přiložena ke kolekci a následně zkopírována pro každý proces. To může vést k velkému zahlcení sítě, ale hlavně budou nadbytečné úkoly vykonávané znova a znova. Uživatel navíc musí tuto hodnotu ze své kolekce vyjmout nebo ji dále zpracovat.

Spark pro takové případy podporuje vytvořením sdílených proměnných typu Broadcast a Accumulator.

1.2.3.1 Broadcast

Proměnná, která je uložena na každém uzlu a je určena pouze pro čtení. Může sloužit například k předání velkého množství dat všem uzlům najednou. V takovém případě lze značně snížit komunikační náročnost. Proměnná je v tomto případě reprezentována dočasným souborem, ke kterému mají všechny výpočetní procesy přístup.

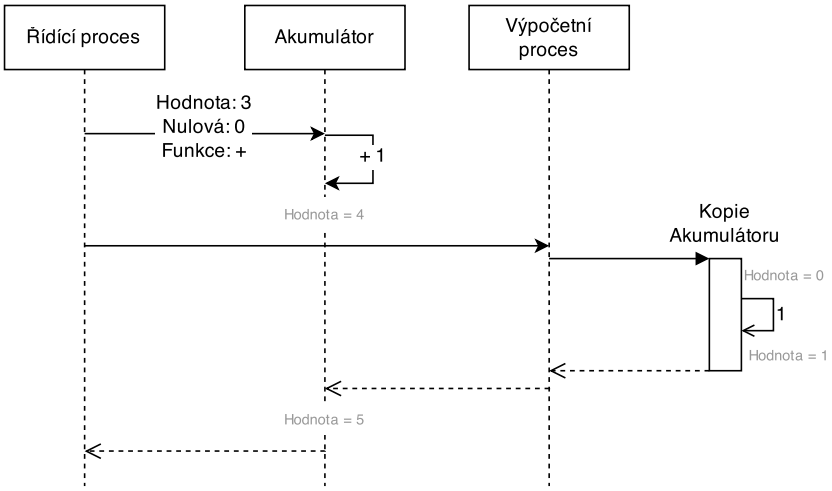
Jakmile je jednou vytvořena, nelze ji měnit. Řídící proces jí může pouze odstranit a vytvořit novou.

```
broadcastVar = sc.broadcast([1, 2, 3])

sc.parallelize([1, 2, 3, 4])
  .map(lambda x: x in broadcastVar.value)
  .collect()

# => [True, True, True, False]
```

1.2.3.2 Accumulator

Akumulátor je proměnná, ke které má výlučný přístup pouze řídící program. Všichni ostatní včetně výpočetních procesů mohou pouze akumulovat (přidávat) data. Před vytvoření této proměnné musí uživatel nejprve definovat funkci, která akumuluje současnou hodnotu a nově přidanou. Taková funkce může být například sčítání, odčítání či hledání maxima.

Při vytvoření je nutné také definovat počáteční „nulovou” hodnotu. Ta se bude posílat do výpočtu místo současné „celkové” hodnoty. Každý výpočetní proces má svou kopii akumulátoru a s ostatními nemůže komunikovat.



Obrázek 1.5: Aktualizace Akumulátoru

Postup:

1. Uživatel definuje hodnotu proměnné (3), počáteční („nulovou") hodnotu pro výpočty (0) a funkci (sčítání)
2. V řídícím procesu lze libovolně hodnotu měnit či akumulovat
3. Do výpočetního procesu je poslána pouze funkce a nulová hodnota
   - výpočetní proces může pouze akumulovat hodnotu ke svojí počáteční
   - zpět je poslána výsledná hodnota (1)
4. Instance akumulátoru akumuluje současnou hodnotu (4) a přijatou (1)

Uživatel si ale musí být vědom, jakou akumulační funkci definuje. Například data ze 2 procesů mohou přijít v různém pořadí a tudíž výsledek nemusí být vždy konzistentní.

```
accum = sc.accumulator(0)

sc.parallelize([1, 2, 3, 4])
  .foreach(lambda x: accum.add(x))

accum.value
# => 10
```

1.3 Mllib

Škálovatelná machine learning knihovna obsahující obecné algoritmy a druhy strojového učení. V následujících kapitolách představím ty nejzajímavější z nich a to základní výpočty, klasifikaci a regresi a shlukovou analýzu. O výpočty se starají 2 knihovny a je na nás, kterou zvolíme.

Pro optimalizované výpočty je použita knihovna `Breeze`, která závisí na `netlib-java`. Tuto knihovnu bych určitě doporučil nainstalovat, protože výpočty se výrazně urychlí. Knihovny jsou implementované ve Fortranu, který dosahuje lepší výsledků pro větší množství dat. `Netlib-java` také podporuje více podpůrných knihoven jako ATLAS, OpenBLAS či Intel MKL.

Pokud není `Breeze` dostupný, použijí se funkce, které jsou součásti Sparku. Ty ale nedosahují takových výsledků jako optimalizované knihovny a tedy můžeme pocítit pokles výkonu.

Strojové učení
Než se pustím do popisu samotných metod, popíši, co to vůbec strojové učení je.

Jsou to algoritmy a techniky, které umožnující stroji učit se. K tomu ale potřebuje zkušební (trénovací) data. Ty obsahují jak konkrétní platné data, tak výsledek, který by měl být vrácen po naučení. Podle zvolené metody učení se pokusí určit vztahy mezi nimi.

Jakmile se stroj naučí „novou věc" tak vrátí model. Model reprezentuje znalost stroje, kterou dále využíváme.

Příklad: Chceme předpovídat počasí na tento den. Máme data z radarů z tohoto týdne a údaje, jaké přitom bylo počasí. Data co máme, vložíme do příslušné metody, která vrátí model. Ten bude obsahovat metodu, která na vstupu dostane aktuální data a na výsledku nám řekne dnešní počasí.

1.3.1 Datové typy

Vektor
Na výběr máme ze dvou typů vektorů. `DenseVector` je takový vektor, který má většinu hodnot nenulových. Naopak pokud máme vektor, kde většina hodnot je nulová, lze použít `SparseVector`. Ten vyžaduje parametry: velikost vektoru a na jakých pozicích budou nenulová čísla.

Obecně je vhodné využívat oba typy kvůli rychlosti operací. Některé operace na `SparseVectoru` mohou být rychlejší, protože nuly bude moci *přeskočit*.

```
denseVector = DenseVector(1, 2, 3)
sparseVector = SparseVector(3, {1 => 2})
```

Oštítkovaný vektor
Vektor obsahující štítek. Spark jej používá pro proces učení, kde štítek představuje finální hodnotu, které by měli data dosáhnout.

```
labeledPoint = LabeledPoint(1, denseVector)
```

Matice
Dvou dimenzionální matice, která stejně jako vektor může být dense ši sparse. Nicméně pro knihovnu mllib se používá pouze `DenseMatrix`, která má vstupní parametry: délka řádku, velikost sloupce a data.

```
matrix = DenseMatrix(2, 2, [1, 2, 3, 4])
```

1.3.2 Základní výpočty

Souhrnná statistika
Souhrnné informace o dat ve sloupcích vektoru. Dostupné metody jsou průměr a odchylka.

```
summary = Statistics.colStats(mat)
summary.mean()
```

Korelace
Korelace mezi dvěma sadami hodnot. Současně jsou podporovány typy Person a Spearman.

```
Statistics.corr(seriesX, seriesY, method="pearson")
```

Vzorkování
Na rozdíl od předchozích metod je vzorkování součásti jádra a je možné ho použít bez mllib knihovny na key-value strukturách. Klíč v tomto případe charakterizuje všechny ostatní hodnoty. Například klíč může být žena/muž a hodnoty věk či výška v dané populaci.

```
data.sampleByKey(False, fractions)
```

1.3.3 Klasifikace a regrese

Klasifikace se zabývá předpovědí z dané množiny dat. Například určení, zda je daný email spam či nikoliv.

Regrese využívá k odhadnutí proměnných další proměně z množiny. Například jde o určení jako bude počasí na základě informací z předešlého dne.

Určení, do jakého typu daný problém spadá, nelze vždy jednoznačně určit a proto jsou některé metody použitelné ve více oblastech. V následujícím textu opět nebudu popisovat všechny, ale pouze ty nejzajímavější.

Metoda nejmenších čtverců
Je statistická lineární metoda operující na soustavě rovnic, kde je více rovnic než počet neznámých. Výsledek je tedy taková hodnota, která má nejmenší odchylku (součet) od ostatních rovnic. Na základě různých regularizací jsou dostupné metody lineární, lasso či ridge.

Ztrátová funkce je definovaná jako

$$L(x, y) := \frac{1}{2}(x - y)^2.$$

Průměrná chyba se potom určuje pomocí

$$\frac{1}{n} \sum_{i=1}^{n} (x_i - y_i)^2$$

Metody: `LinearRegressionWithSGD`, `LassoWithSGD`, `RidgeRegressionWithSGD`.

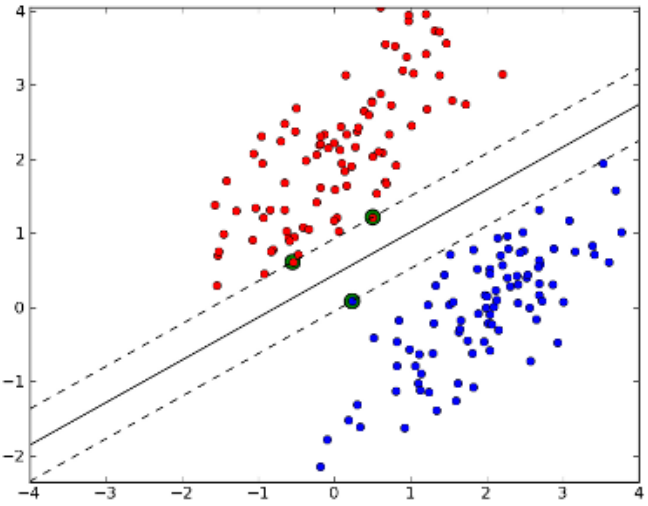
Logistická regrese
Odhaduje pravděpodobnost výskytu jevu na základě předchozích proměnných, které mohou daný jev ovlivnit. Ztrátová funkce v tomto případě vypadá následovně.

$$L(x, y) := \log(1 + \exp(-yx)).$$

Metody: `LogisticRegressionWithLBFGS`, `LogisticRegressionWithSGD`.

Support vector machine
Metoda strojového učení hledající nadrovinu, která optimálně odděluje data. Optimálně znamená, aby průměr vzdáleností všech bodů od roviny byl co největší. Na vytvoření této roviny stačí pouze body, které jsou nejblíž. Ty se pak nazývají „podpůrné body".

Metody: `SVMWithSGD`.



Obrázek 1.6: Support Vector Machine

1.3.4 Shluková analýza

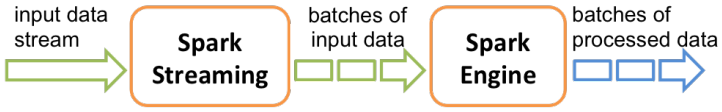
Statistická metoda sloužící k seskupování hodnot do jednotlivých shluků (skupin) tak, aby si prvky v jedné skupině byly co nejpodobnější a zároveň, aby se co nejvíce odlišovaly od ostatních.

K-nejbližších sousedů
Jedna z nejznámějších a nejvíce používaných metod pro shlukovou analýzu.

Metody: `KMeans`.

1.4 Streaming

Rozšíření, umožňující vytvořit škálovatelný, vysoce propustný a chybově odolný proud, který bude v reálném čase zpracovávat data. Výsledky mohou být uloženy na lokálním souborovém systému nebo několika typech databází (například Cassandra).



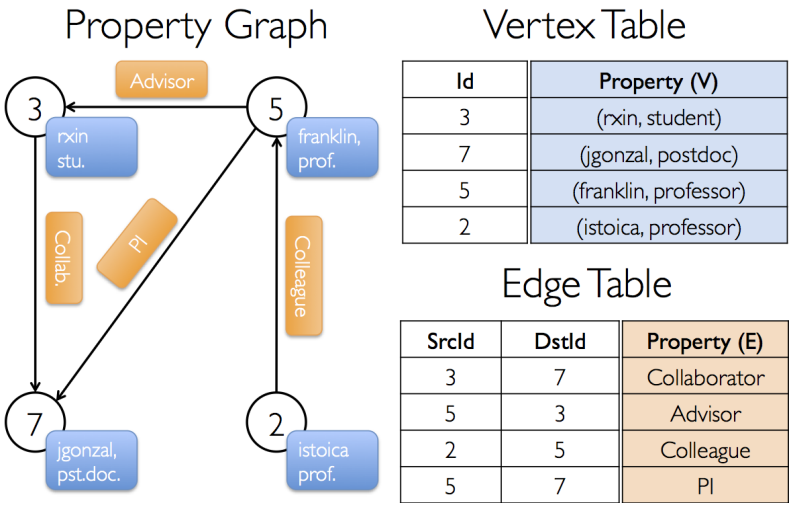
Obrázek 1.7: Spark streaming

1.5 SQL

Umožňuje strukturovat a zpracovávat data z databáze pomocí SQL. Data jsou uložena a dostupná ze třídy `DataFrame`

1.6 GraphX

Obsahuje metody a postupy pro zpracovávání grafových algoritmů. Graf je uložen jako množina vlastností hran a uzlů, které jsou propojené. Do tohoto typu je možné vložit takřka jakýkoliv graf.



Obrázek 1.8: Vlastní graf

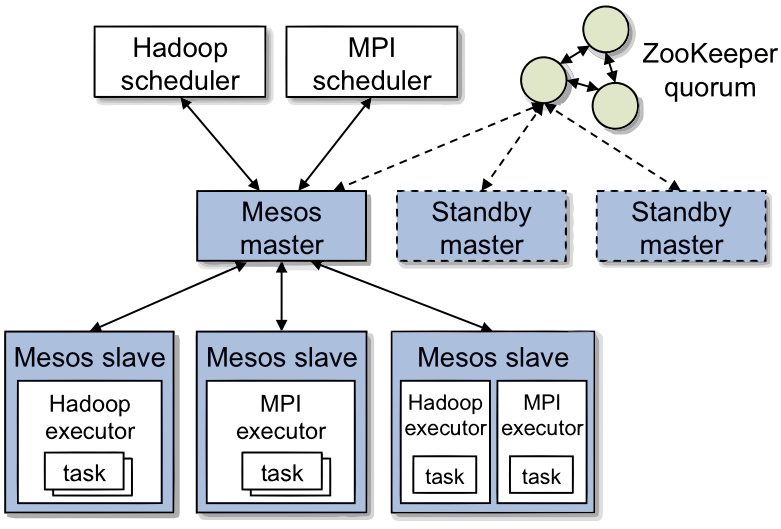
Kapitola 2

Správa clusteru

2.1 Mesos

Je open-source cluster manager vyvíjený univerzitou Berkeley. Implementovaný je v jazyce C++ a šířen pod svobodnou licencí Apache. Mesos pracuje mezi kernel a aplikační vrstvou a poskytuje interface pro jazyky Java, Python a C++.

Má za úkol oddělit prostředky výpočetních uzlů a poskytovat jejich abstrakci přes celou sít. Tato sít je škálovatelná do 10 000 uzlů a pracuje na bázi *master-slave*. Zároveň je také odolná vůči výpadkům a chybám díky použití systému *ZooKeeper*. Ten v případě chyby na hlavním (řídícím) uzlu zvolí nový hlavní uzel.



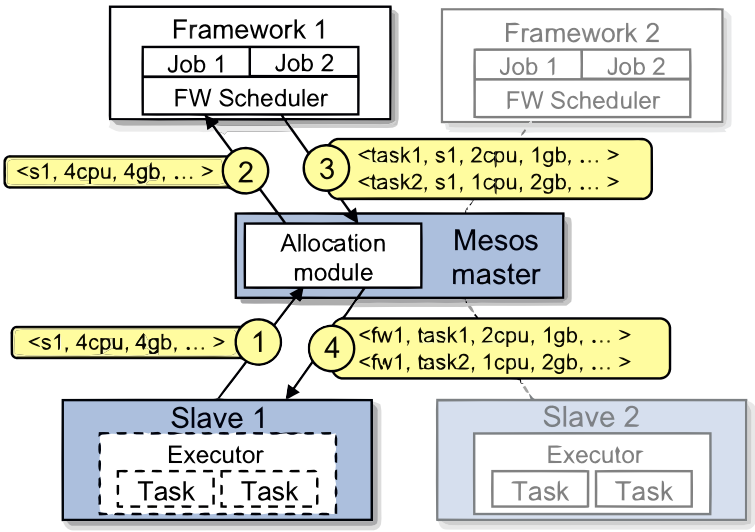
Obrázek 2.1: Architekrura Apache Mesos

Na začátku jsou výpočetní prostředky zaregistrovány a následně přidělovány. Mesos může alokovat takřka jakékoliv výpočetní prostředky včetně CPU, paměti nebo disku.

Na obrázku 2.1 vidíme hlavní komponenty. Mesos se skládá z hlavního uzlu (master), který spravuje všechny ostatní (slave). Na těchto uzlech běží aplikace, které se nazývají framework.

Master umožňuje sdílení prostředků přes více aplikací na více uzlech. Každému frameworku jsou nabízeny prostředky a je na ní, zda je příjme. Master vždy rozhoduje, kolik výpočetních prostředků jaké aplikace dostane. Tento počet se může určovat podle mnoha politik. Například pomocí *fair-sharing*, kdy všechny aplikace dostanou prostředky spravedlivě. Framework si následně vybírá a rezervuje ty, které potřebuje.

Každý framework se skládá ze dvou komponent. Registr, který komunikuje s masterem ohledně nabízených prostředků a konkrétní úkoly. Registr také vybírá konkrétní prostředky z nabídky. Následující obrázek 2.2 demonstruje mechanismus nabídky. Každý framework obsahuje jeden registr, ale výpočetních úloch může mít více. Navíc každá může mít jiné požadavky.



Obrázek 2.2: Nabídky v Apache Mesos

1. **Slave 1** oznámí masterovi, že má 4 CPU a 4 GB paměti. **Master** podle zvolené politiky vyhodnotí, že všechny tyto prostředky by měl dostat **framework 1**.

2. Master pošle nabídku **frameworku 1**, která obsahuje popis volných prostředků.

3. **Registr** pro daný framework odpoví, že spravuje 2 úkoly, které potřebují <2 CPU, 1GB paměti> a <1 CPU, 2GB paměti>.

4. Nakonec pošle **master** oba tyto úkoly na daný uzel, kde jim místní **executor** přidělí prostředky a spustí je. Jelikož úkoly nevyčerpaly všechny dostupné prostředky, **master** je může dále nabízet.

2.2 Hadoop YARN

Apache Hadoop je open-source aplikace napsané v Javě, sloužící pro skladování a zpracovávání velkého množství dat, které jsou uložené na mnoha strojích spojených do jednoho clusteru.

Tato aplikace se skládá:

- **Hadoop Common**: nezbytné knihovny a aplikační moduly
- **Hadoop Distributed File System (HDFS)**: distribuovaní souborový systém
- **Hadoop YARN**: správce výpočetních prostředků v clusteru
- **Hadoop MapReduce**: programový model pro zpracovávání dat

Pro nás je v tuto chvíli nejzajímavější pouze část **Hadoop YARN**.

Je to systém na správu prostředků, uzlů a a uživatelů. Skládá se z **ResourceManager**, který rozhoduje o všem ohledně prostředků a **NodeManager**, který se stará o jednotlivý uzel.

ResourceManager je v podstatě jenom plánovač. Jeho funkcionalita je omezena na vyjednávání prostředků mezi jednotlivými aplikacemi. Vyjednání stejně jako v případě Mesosu probíhá na základě nějaké politiky. ResourceManager má také mnoho zásuvných pluginů, které mění algoritmus přidělování prostředků.

ApplicationMaster koncept je novinkou pro veze *2.x*, který má za úkol vyjednávat s ResourceManagerem a zároveň se pracovat s **NodeManagerem**, kde bude úlohy vykonávat, monitorovat používání prostředků a sledovat postup.

Navíc tato komponenta obsahuje většinu vlastností tradičního pojetí **ResourceManagera**, takže celý cluster dosahuje velmi dobré škálovatelnosti, kdy lze bez větších problémů propojit 10 000 uzlů. Proto je **ResourceManager** také navržen pouze jako plánovač, který neodpovídá za problémy v síti.

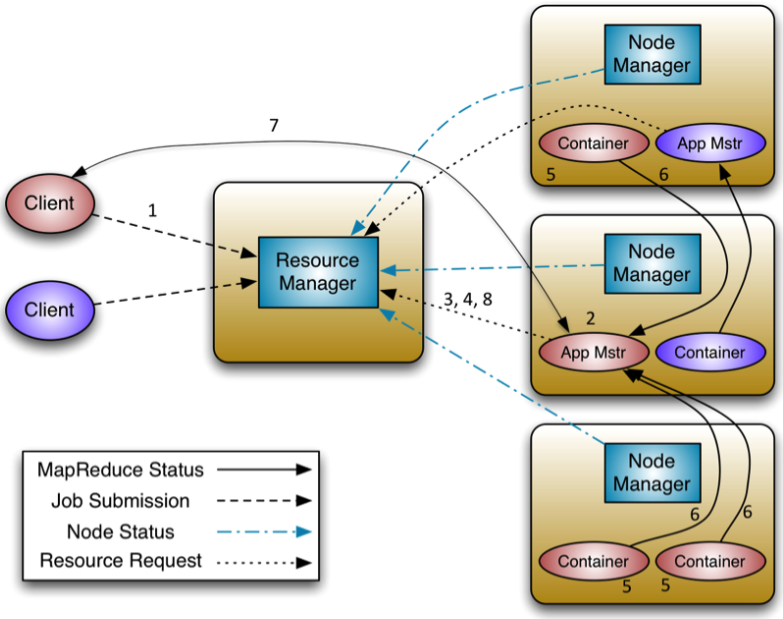
Na rozdíl od Mesosu nepotřebuje tento cluster *ZooKeeper*, který by udržoval konzistentní sít. To ale ani není potřeba, jelikož **ApplicationMaster** je definovaný pro každou aplikaci.

Model výpočetních prostředků je velice obecný a dovoluje aplikaci žádat o velmi specifické prostředky jako:

- hostname, rackname
- CPU, paměť
- disk, síť, GPU
- ... mnoho dalšího

ResourceRequest je specifický požadavek na prostředek ve tvaru `(jméno, priorita, požadavky, počet kontejnerů)`. Tyto požadavky jsou důležité hlavně proto, že každá aplikace může skrz svého mastera požádat o sdílení prostředků.

Kontejner (Container) je prostředek přidělený aplikaci a schválený **ResourceManagerem**. Dává aplikaci právo použití konkrétního prostředku na daném uzlu.



Obrázek 2.3: Proces spuštění aplikace na Apache YARN

1. Klient nahraje aplikaci s nezbytnými parametry a tím je vytvořen **ApplicationMaster**.
2. **ResourceManager** vyjedná specifický **Container**, ve kterém aplikace poběží. Na daném uzlu je následně spuštěna.
3. **ApplicationMaster** se při načítání zaregistruje.
4. **ApplicationMaster** vyjednává o příslušných prostředcích.
5. Po úspěšné alokaci jsou kontejnery spuštěny podle specifikace dané **NodeManagerem**.
6. Aplikace poskytuje nezbytné informace (postup, status, ...) ke svému **ApplicationMaster** přes specifický protokol.
7. Jakmile je aplikace hotová, **ApplicationMaster** zruší registrace prostředků a ukončí se.

Kapitola 3

Ruby

Diplomová práce se nezabývá detailní analýzou tohoto jazyka. Nicméně pro pochopení další kapitoly je nutné vědět alespoň základy.

Ruby je obecně použitelný skriptovací jazyk vytvořen Yukihirem Matsumotem v roce 1990. Původní návrh vycházel z jazyku Perl, Smalltalk a Lisp. Dnešní verze se však od nich velice odlišuje. Nynější verzi bych co do podobnosti asi nejvíce přirovnal k Pythonu. Tento jazyk je navíc již ve Sparku přítomen a tak bude zajímavé jejich výkonové porovnání.

> *Ruby je jednoduchý ve vzhledu ale komplexní uvnitř, stejně jako člověk.*

Největší přednost či pro někoho největší nevýhoda je jeho pojednání objektového programování. Java je například také objektový jazyk, nicméně zde nalezneme primitivní typy jako Int či Long. V Ruby toto není, protože vše co se vytvoří je objekt. Dokonce i definice objektu je objekt.

Existuje mnoho implementaci. Nicméně původní a oficiální je v jazyce C. Označeno jako **MRI**. Ostatní implementace se snaží být co nejvíce kompatibilní, ale ne vždy je to možné. Běžných funkcí se tento problém netýká, při použití těch specializovaných může být rozdíl značný.

- Ruby je oficiální implementace v C.
- JRuby je postaven na jazyce Java. Obsahuje rozdílnou implementaci GC, vláken a několika dalších knihoven.
- Rubinius je zvláštní implementace Ruby v Ruby.
- MRuby je odlehčená implementace oficiálního Ruby, která se dá připojit k jiným knihovnám.
- IronRuby je implementace v .NET frameworku.

3.1 Balíčky

Oproti jiným jazykům zde nenalezneme tradiční pojetí knihoven. V Ruby to jsou gemy (balíčky). Každý gem obsahuje definici a samotný kód. Definice může obsahovat:

- informace o autorovi
- jméno gemu
- popis gemu
- systémové závislosti
- závislosti na ostatních balíčcích
- ...

Tyto gemy lze navíc zveřejnit celé komunitě přes RubyGems. Obdobný princip najdeme v Javě za použití například balíčku a Maven repozitáře.

3.2 Funkce a metody

V Ruby lze vytvořit jak „klasické" metody, tak ty anonymní. Anonymní funkce je taková funkce, která nemá jméno a lze ji uložit do proměnné či jí poslat parametrem. Navíc v paměti existuje jen omezenou dobu (stejně jako ostatní proměnné).

Příklad definice a volání klasické metody:

```
def suma(x, y)
  x + y
end

suma(1, 2)
# => 3
```

Příklad anonymní funkce:

```
suma = lambda do |x, y|
  x + y
end

suma.call(1, 2)
# => 3
```

Jediný rozdíl v použití je, že metodu lze volat přímo, kdežto tu anonymní za použití `.call()`. Ruby navíc obsahuje techniky jak z klasické udělat anonymní.

3.3 Garbage collector

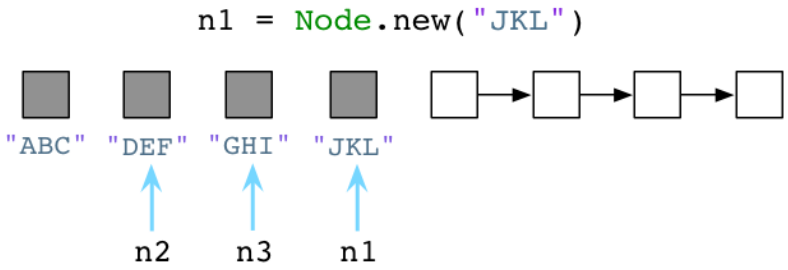
Je jednoduše řečeno **správce paměti**. Tato oblast je zařazena pouze, abych čtenáři dal malý náhled do problematik, které jsou zmíněny v implementační kapitole. Popsány jsou základní věci pro verzi 2.

GC má za úkol:

- alokovat paměť
- ukládat informace o objektech
- vracet paměť zpět

3.3.1 Jak funguje

GC v Ruby pracuje na principu *Mark and Sweep*. Na začátku si alokuje paměť, která je rozdělena na menší dílky o stejné velikosti. Ty jsou zřetězeny pomocí seznamu. Jakmile přijde požadavek na paměť, tak se použije první prázdný dílek.



Obrázek 3.1: Ruby Garbage Collector

Například Python na to jde jinak. Na začátku nemá žádný seznam a paměť si alokuje postupně. Jakmile objekt zanikne, je tato část vrácena zpět.

To je veliký rozdíl oproti Ruby. Ten si nejen na začátku alokuje paměť, kterou nemusí nikdy použít, ale po uvolnění si ji ponechává pro budoucí použití. V některých situací to může představovat problém, protože ačkoliv kód již tolik paměti nepotřebuje, stále je k dispozici. Na druhou stranu tento přístup zrychluje práci GC. Na rozdíl od Pythonu, který má všechny alokace v pořádku a má jen to co potřebuje, Ruby se nemusí tak často starat o jejich uklízení.

3.4 Výkonnostní porovnání

Porovnání je vzato z The Computer Language Benchmarks Game. Výsledky nelze brát jako závazné, jelikož to jsou časy nejlepších dostupných algoritmů pro daný jazyk.

- Ruby 2.1
- Python 3.4

Test	**Ruby**	**Python**
Binární stromy	60s	129s
Spektrální norma	102s	211s
Problém n těles	673s	904s
Číslo π	11s	2s
Mandelbrotova množina	494s	239s
FASTA	158s	96s

Následující tabulka obsahuje výsledky vytvoření datových struktur. Tyto časy jsou více závazné, neboť jsou implementačně nezávislé.

- Ruby 2.2.1
- Python 3.4.2
- Pole měli velikost 10 000 000

Test	**Ruby**	**Python**
Pole obsahující 1	0.6s	1.4s
Pole (hodnota odpovídá indexu)	0.6s	1.4s
Pole náhodných hodnot (float)	0.9s	1.6s
Pole náhodných ascii (char)	2.8s	2.9s
Pole textu (string)	4.2s	31s
Vytvoření a součet (int)	1.3s	1.7s
Vytvoření a součet náhodných hodnot (float)	1.5s	3.1s

Na první pohled se zdá, že Ruby ve všem předčí Python. Na druhou stranu zde byl veliký rozdíl v použití paměti. Python po „sobě uklidil" po každém testu a na ten nejnáročnější potřeboval 200MB. Ruby si paměť vždy nechal a na celé vyhodnocení potřeboval 600MB. To je důsledek různě implementovaného Garbage collectoru.

Bohužel nemohu doplnit údaje, kolik každý test potřeboval, protože Ruby nedovoluje manuální vyčištění paměti.

Kapitola 4

Návrh a implementace

V předchozích kapitolách byly popsány technologie Apache Spark, správy clusteru a jazyk Ruby. Nyní vystává otázka *„jak je propojit?"*. Naštěstí již existovaly projekty, které se podobnou otázkou zabývaly a já se mohl nechat inspirovat.

První byla samozřejmě implementace v Pythonu, který je součásti vývoje celého balíčku Spark. To ovšem představovalo menší problém, protože tento jazyk měl podporu přímo v jádře. Druhá implementace byla v jazyce R, který už byl mimo jádro. Ačkoliv jsou tyto dva jazyky od Ruby dosti vzdálené, tak jejich pojetí implementace mi na začátku velice pomohlo. Tehdy bylo nutné určit jakési přemostění *Ruby-Spark-Ruby* neboli *Řídící program-Spark-Výpočet*.

Postup implementace je popsán v následujících kapitolách, kde je navíc srovnání s ostatními jazyky. Aby byl výpočet možný, bylo třeba vyřešit tyto problémy.

- Propojení Ruby-Java
- Serializace dat
- Definice funkcí
- Definice úkolu
- Výpočet

4.1 Propojení Ruby-Java

Před jakoukoliv implementaci bylo třeba vyřešit relativně složitý problém *„lze a jak volat Javu z Ruby?"*. Z předchozí kapitoly víme, že Ruby je také implementován v Javě (JRuby). V JRuby je dokonce velice jednoduché připojit Javovské knihovny a pracovat s nimi jako s těmi z Ruby. První verze mé diplomové práce také pracovala jenom s JRuby, nicméně cílem bylo vytvořit více variabilní knihovnu, které v prvé řadě podporuje oficiální C interpret MRI.

Naštěstí je k dispozici knihovna, která to umožňuje. RubySpark nakonec obsahuje adaptéry, které umožňují práci jak na MRI, tak na JRuby. Implementace navíc obsahuje takové postupy, díky kterým nemusí uživatel tuto otázku řešit.

4.1.1 RJB

Neboli *Ruby-Java Bridge* vytváří most mezi Ruby a Javou. Na pozadí je spuštěn JVM dle zvolených parametrů, se kterým komunikuje. Je to vlastně JNI rozhraní. V Ruby jsou následně vytvořeny abstrakce Javovských tříd, které lze normálně používat.

Programátor musí mít neustále na paměti, že daná instance se nachází na JVM. RJB sice nabízí přímý převod základních typů, nicméně u složitějších datových struktur je nutný vlastní převod.

Problémy na které jsem narazil:

- Všechny metody musí být volané přesně s danými typy. Pokud například funkce vyžaduje jako parametr typ `Long` a já pošlu `Int` nebo místo 3 parametrů 2, tak RJB vyhodí výjimku „chybějící metoda".
- Pole vnořené v poli (více dimenzionální) se interpretuje jako jednorozměrné pole.
- RJB neumí správně převést pole s typem `Boolean`.
- Nelze jednoduše používat typové šablony.

4.1.2 JRuby

V implementaci JRuby je podpora Javy vestavěna, takže odpadlo mnoho starostí. Jelikož lze RubySpark použít i za pomocí RJB, bylo nutné zajistit konzistentní chování. To bohužel nebylo vždy možné a v implementaci se nachází pár podmínek, které upravují chování na základě zvoleného adaptéru.

Problémy s JRuby:

- Pole objektu s vrací zpět jako pole `Byte`.
- Všechny čísla se převádí na typ `Long`.

4.2 Serializace dat

Po úspěšném implementování adaptéru mezi Javou a Ruby bylo nutné vytvoření mechanismu, jak do Sparku dostat data. Jediný způsob je data serializovat za pomocí připravených knihoven a výsledek poslat dále. Během implementace jsem vyzkoušel knihovny Marshal, YAML, MessagePack, Oj, Ox a JSON. Některé se ukázaly jako nepoužitelné a tak nakonec zvítězily pouze Marshal a Oj.

Pro výpočet obvykle stačí pouze jeden serializační nástroj, ale v některých případech je žádoucí mít dva. Například pokud uživatel vloží data ve formě textového souboru. V takovém případě musí výpočetní proces nahrát data jako text (s příslušným kódováním). Nad daty mohou být provedeny různé transformace, které vytvoří takové struktury, které nelze uložit jako prostý text. Kvůli takovýmto situacím existuje **serializator** a **deserializator**.

Serializator
Použitý ve výpočetním procesu k serializování výsledných dat a následně v řídícím procesu k jejich opětovnému sestavení.

Deserializator
Využitý pouze ve výpočetním procesu k načtení příchozích dat.

4.2.1 Druhy

Všechny serializační ukázky v této kapitole jsou výsledek této datové struktury.

```
data = [1,2,3]
```

Marshal
Binární serializace, která je navíc součást jádra Ruby a dostupná pro všechny verze a implementace. Pro to je zvolena jako výchozí. Mezi jeho největší přednosti patří možnost serializovat takřka vše. Od obyčejných polí po uživatelsky definované struktury. První 2 byty vždy označují verzi. Následují binární značky struktury.

```
"\x04\b[\bi\x06i\ai\b"
```

JSON
Navržen jako univerzální datový zápis bez ohledu na programovací platformu. Určený hlavně k přenosu dat a dnes jej nalezneme nejčastěji u webové komunikace. Výstup je vždy textový řetězec.

```
"[1,2,3]"
```

YAML
Mezi jeho největší přednosti patří fakt, že je velice dobře čitelný člověkem. Inspirovaný je formátem XML, ale bez tagového ohraničení. Bohužel mezi stinné stránky patři velice pomalá rychlost zpracování.

```
---
  - 1
  - 2
  - 3
```

MessagePack
Stejně jako Marshal je tato serializace binární. Vychází z JSON, ale pro reprezentaci datových tříd používá binární značky. V obou srovnáních je navíc jasným vítězem a jeho verze je dostupná jak pro MRI, tak JRuby. Bohužel má jednu zásadní vadu. Nedokáže serializovat velká čísla a uživatelsky definované třídy. Nicméně kvůli ostatním výhodám je implementována jeho podpora a lze jej použít.

```
"\x93\x01\x02\x03"
```

Oj
JSON serializer ale mnohem rychlejší. Stejně jako Marshal dokáže serializovat takřka všechno a proto jej nejvíce doporučuji. Ale je dostupný pouze pro MRI a tak se nemohl stát výchozím nástrojem.

```
"[1,2,3]"
```

Ox
Jako jediný ze zmíněných nástrojů využívá XML formátu. Vlastnosti má podobné jako Oj, bohužel díky definici XML má serializace velkou velikost a proto není pro RubySpark použitelný.

```
<a>
  <i>1</i>
  <i>2</i>
  <i>3</i>
</a>
```

4.2.1.1 Srovnání

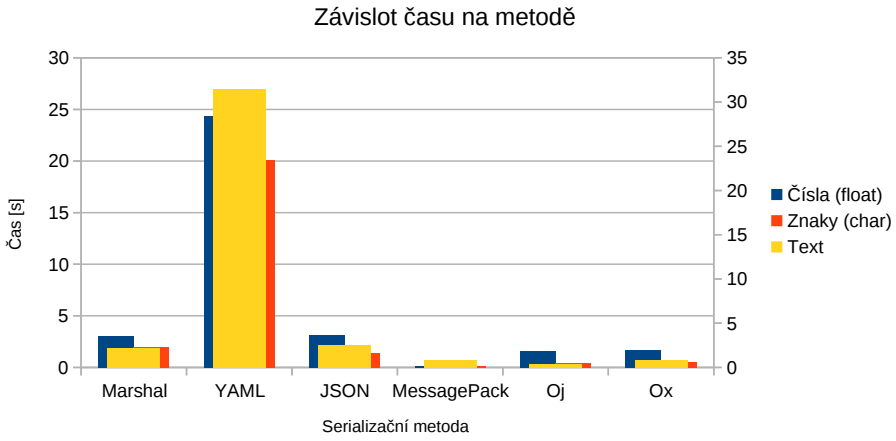
Všechny výše zmíněné serializační techniky jsem porovnal na třech testech.

Čísla (float) vytvoření pole o velikosti 1 000 000, které obsahuje náhodná desetinná čísla.

Znaky (char) opět vytvoření pole o velikosti 1 000 000, které ale nyní obsahuje pouze znaky z ASCII tabulky.

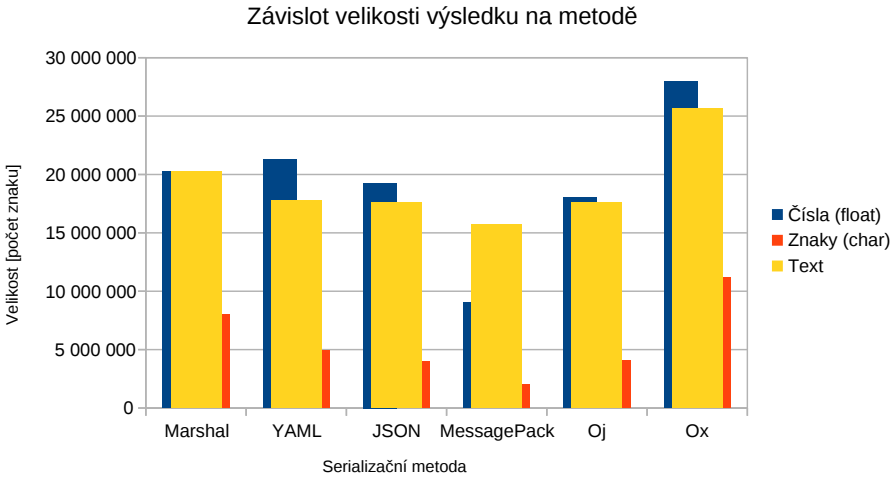
Text pole, kde každá položka obsahuje text o velikosti 20ti znaků.

Na obrázku 4.1 vidíme závislost času na zvolené metodě. Čas obsahuje jak serializaci datové struktury, tak její opětovnou deserializaci. MessagePack v tomto testu jednoznačně vyhrál. Zato YAML je nejpomalejší.



Obrázek 4.1: Srovnání časové závislosti různých serializačních technik

Na dalším obrázku je vidět, jaká byla velikost serializovaných dat. Stejně jako v předešlém grafu je i tady vítěz MessagePack. Naopak nejhůř dopadl Ox kvůli svému XML formátu.



Obrázek 4.2: Srovnání velikosti výsledku pro různé serializační techniky

4.2.2 Předání

Jelikož most do Javy byl hotov a funkční, mohl jsem ze začátku předávat data jako parametr funkce. To se ale ukázalo jako velice náročné a pomalé.

Pro řešení tohoto problému jsem se inspiroval u implementace Pythonu, který narazil na stejný problém. Ten na začátku data uložil do binárního souboru, který si pak Spark načetl. Tento způsob se velice osvědčil a byl implementován. Jedinou stinnou stránkou představuje redundance dat. Originální data existují v RubySpark, další je na disku a poslední kopie je ve Sparku.

Tento problém trvá naštěstí jen krátkou dobu, protože originální data budou odstraněny Garbage Collectorem v Ruby a kopie na disku je odstraněna hned po úspěšném načtením.

4.3 Definice funkcí

Dále bylo třeba vyřešit, jak by uživatel mohl definovat funkci, kterou bude možné načíst a odeslat. Na počátku se zdálo, že to nebude obtížný úkol. Později se ukázalo, že je to velký problém, který není dosud vyřešen podle mých představ. Nejprve ale předvedu můj prvotní záměr. V Ruby lze udělat následující:

```
nums = 3
data = [1,2,3]
func = lambda{|x| nums*x*2}

data.map{|x| nums*x*2}
data.map(&func)

# => [6,12,18]
```

Výsledek je v obou případech stejný. Stejného použití mělo být docíleno i pro Spark metody. Pro lokální počítání zde problém není. Ten nastane v případě, když chci tyto anonymní funkce serializovat. V Ruby totiž nelze udělat:

```
func = lambda{|x| x*2}
Marshal.dump(func)
```

To je dané tím, že anonymní funkce jsou v Ruby označeny jako *uzávěry (clousures)*. Tyto funkce, stejně jako proměnné, existují v paměti. Na rozdíl od nich je k nim ale připojena celá paměť (jednoduše řečeno). Tím pádem lze v takové funkci použít jiné lokální metody, data nebo instance. To je sice obrovská výhoda, nicméně to znamená, že spolu s funkcí je nutno serializovat celou paměť.

Tento problém je vyřešen částečně za pomocí knihovny Sourcify a metody `.bind()`. Ruby sice neumí serializovat Proc, zato ale má uloženou informaci, kde byl vytvořen. Například pokud bude uložena v souboru, lze soubor načíst a podle příslušného řádku definici vyjmout. Přesně na tomto principu pracuje tato knihovna. Použití je ale omezeno následujícím seznamem.

- Funkce musí být definovaná obvyklým způsobem a ne pomocí `.eval()`. V tomto případě definice neexistuje.
- Na jednom řádku lze mít pouze 1 funkci. Při čtení není možné určit, která je správná.
- Serializuje se pouze definice a ne přidružená paměť. Na prvním příkladu by tedy nešlo funkci použit znova, protože proměnná `nums` nebude existovat.
- Nelze použít nejnovější notaci `->{}`.

Tyto nedostatky musí mít uživatel na paměti a neustále s nimi počítat. Třetí bod se dá částečně eliminovat. O tom ale bude řeč až v kapitole příklady použití.

4.4 Definice úkolu

V této kapitole je popsána nejdůležitější vlastnost pro uživatele a to jak definovat výpočet. V kapitole 1.2.1.2 byla popsána ukázka výpočtu počtu slov Python. Kód vypadal takto:

```
file = spark.textFile("hdfs://...")

file.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a+b)
```

Stejný úkol v Ruby:

```
file = spark.text_file("hdfs://...")

file.flat_map(lambda{|line| line.split})
    .map(lambda{|word| [word, 1]})
    .reduce_by_key(lambda{|a, b| a+b})
```

Kromě menšího rozdílu v syntaxi lambdy je další rozdíl v zápisu metod. Python stejně jako Scala využívá *lower camel case* zápis, kde první znak je malý a všechny další slova začínají velkým písmenem. Ruby ale má jinou filosofii zápisu a proto jsou jména oddělené podtržítkem. Nicméně, aby byla zajištěna co největší kompatibilita, jsou dostupné oba zápisy. Například `.flat_map` je ekvivalentní s `.flatMap`.

V dalších kapitolách popíšu, jak se definice vyvíjely. Ty budu demonstrovat na jednoduchém příkladu, který je pro všechny stejný.

Úkol Vybrat všechny slova z kolekce, které mají více než 3 znaky.

Kolekce

```
["a", "b", "c", "ruby", "spark"]
```

Definice

```
spark.map(lambda{|work| [word, word.size]})
     .filter(lambda{|word, size| size > 3})
     .map(lambda{|word, size| word})
```

Nejprve si ke každému slovu přidám hodnotu o jeho velikosti, následně celou kolekci vyfiltruji a nakonec opět vrátím pouze slova. Tento příklad se dá napsat mnohem efektivněji, nicméně to není cílem této ukázky.

4.4.1 První verze

Na začátku byla implementace velmi naivní, kdy se mohla odesílat pouze jedna definice. Výsledek byl jeden velký text (Ruby kód), který se před výpočtem vykonal

```
"@_f = lambda{|work| [word, word.size]};
    Proc.new{|data| @_f.call(data)}"
```

Tato funkce se volala nad každou položkou kolekce.

4.4.2 Druhá verze

Další již umožňovala vypočítat celý příklad. Jako parametr dostávala celou kolekci. Nicméně stále to byl jen text, který se s větším počtem metod stával velice nepřehledný. Navíc stoupalo riziko, že se do textu dostane něco, co by nemělo.

```
"@_function_ = lambda{|work| [word, word.size]};
    Proc.new {|_, iterator| iterator.map{|i| @_function_.call(i)} }"
"@_function_ = lambda{|word, size| size > 3};
    Proc.new {|_, iterator| iterator.filter{|i| @_function_.call(i)} }"
"@_function_ = lambda{|word, size| word};
    Proc.new {|_, iterator| iterator.map{|i| @_function_.call(i)} }"
```

Všechny funkce se vykonávaly postupně. Tento přístup jsem mnohokrát vylepšil, nicméně stále to byl pouze text. Metody `.map` a `.filter` jsou navíc relativně jednoduché. Problém ale nastal u těch složitějších. Například pro `.histogram` je tento postup značně nevyhovující. Navíc do zápisu nemohly být jednoduše přidány žádné další věci jako proměnné či parametry.

4.4.3 Konečná verze

Na začátku byla jednoduchá úvaha: „proč posílat několikrát stejný text, když dělá to samé?". V tomto případě mám na mysli:

```
Proc.new {|_, iterator| iterator.map{|i| @_function_.call(i)} }
```

Stačilo by poslat jen nějakou referenci či lépe třídu, které bude vědět co dělat. Navíc by se to shodovalo s filosofii Ruby *„všechno je třída"*. Proto jsem začal pátrat, jestli existuje nějaká technika řešící daný problém. Našel jsem návrhový vzor **Command**.

> Command zapouzdřuje operace a jejich parametry tak, aby šly později volat. Uživatel vytvoří sadu příkazů, které vloží do vzoru. Jakmile jsou všechny operace vložené, lze toto zapouzdření volat s tím, že se všechny operace budou volat v takovém pořadí, v jakém byly definovány.

Na konec má tedy každá operace v RubySpark svoji třídu, která definuje její chování. Jediné co potřebují, jsou hodnoty od uživatele. Tedy například `.map` vlastní `MapCommand`, který jako parametr vyžaduje funkci, která bude transformovat každou položku. Výsledek bude nyní (velice zjednodušeně) vypadat takto:

```
Command: [
  MapCommand(lambda{|work| [word, word.size]}),
  FilterCommand(lambda{|word, size| size > 3}),
  MapCommand(lambda{|word, size| word})
]
```

Tento postup je velice snadno rozšířitelný, přehledný a do výpočtu skutečně posílá jen to nejnutnější. Co se tedy děje, když definujeme operaci nyní?

1. Kolekce již obsahuje `CommandBuilder`?

   ano: dojde ke zkopírování

   ne: vytvoření nového

2. Parametry a jméno operace jsou poslané dále k sestavení.

3. Parametry jsou podle definice otestovány (zda jsou validní).

4. Všechny potřebné proměnné jsou převedeny na požadovaný tvar (například serializace anonymních funkcí).

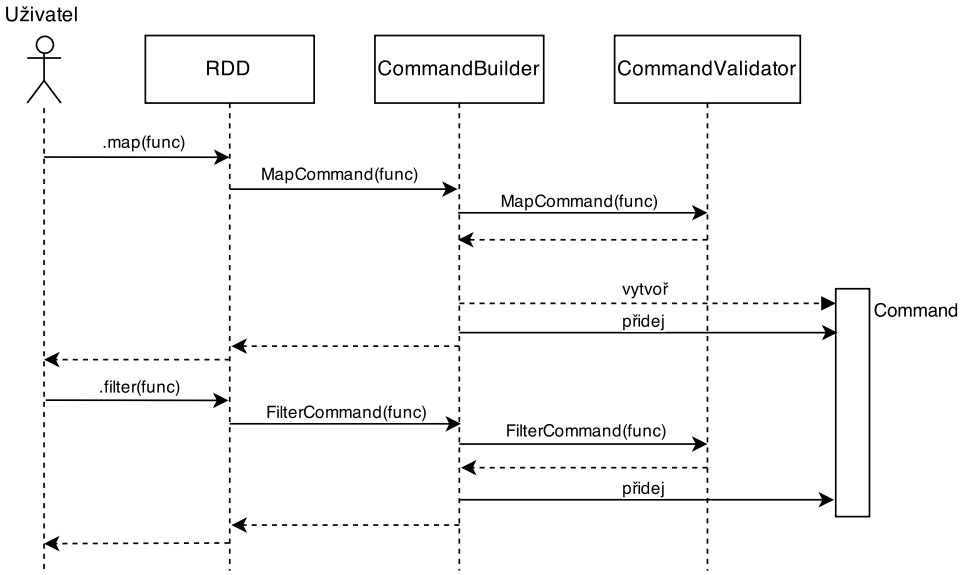
5. Výsledek je třída, které se vloží do seznamu (zapouzdření).

Proč se vůbec kopíruje?

V prvním bodu se testuje, zda `CommandBuilder` již existuje a když ano, dojde k jeho kopírování. To je dané možností řetězit příkazy ve Sparku. Tato vlastnost byla popsána v odstavci 1.2.1. Nejednodušší bude asi ukázka. Představme si, že máme 2 transformace. První násobí čísla dvěma a druhá třemi.

```
rdd1 = spark.map(lambda{|x| x*2})
rdd2 = rdd1.map(lambda{|x| x*3})
```

Pokud by při vytváření druhé transformace nedošlo ke zkopírování, tak by se změna projevila i u první. V takovém případě by již nebyla použitelná a obě by se rovnaly.



Obrázek 4.3: Postup definování příkazu pro map a filter

Popis

`Command` zapouzdření obsahující operace a další informace. Tato třída se serializuje a posílá do výpočetního procesu.

`CommandBuilder` se stará o správné sestavení `Command`.

`CommandValidator` testuje platnost hodnot před uložením.

`*Command` obsahují kód, který definuje konkrétní operace (mapování, řazení, . . .).

Další výhody

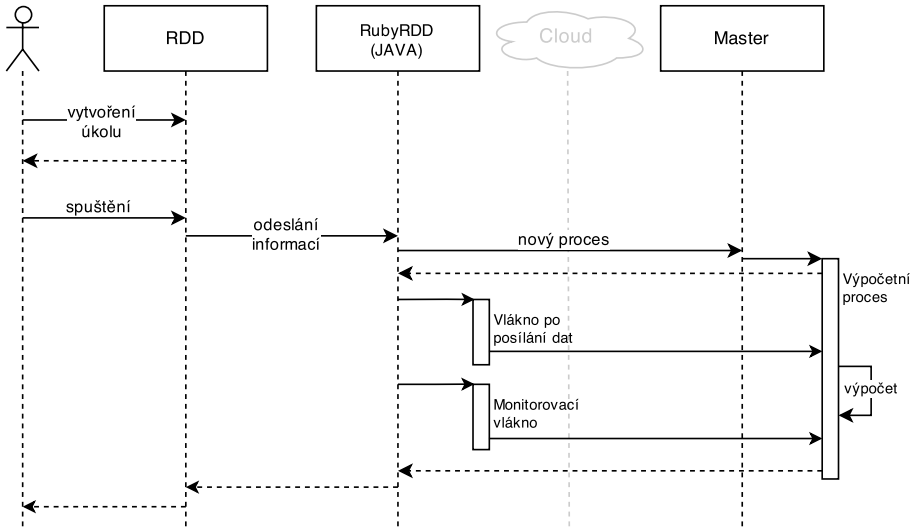
Jelikož se do výpočtu již neposílá jeden velký text ale třída, lze do ní také přidat další parametry upravující chování či přidávají funkcionalitu.

Třída `Command` obsahuje následující:

- serializer a deserializer (více informací v kapitole 4.2)
- zřetězené operace
- Ruby knihovny nutné pro výpočet (pouze názvy)
- vázané objekty (popis v sekci 5.5)

4.5 Výpočet

Nyní už máme vše potřebné k výpočtu. Máme propojení do Javy, serializovaná data, nadefinované funkce i připraveno zapouzdření všech informací. Vše je tedy připraveno k výpočtu.



Obrázek 4.4: Vypočítání úlohy

1. Nejprve je nutné nahrát data a definovat úkol.
2. Jakmile je úkol připravený, můžeme spustit jeho výpočet. To se dělá pomocí příkazu `.collect()`.
3. Do Javovského objektu RubyRDD se pošlou nezbytné informace.
4. Je zajištěna existence Master procesu, který vytváří výpočetní procesy.
5. Master vytvoří proces starající se o výpočet.
6. Jakmile je proces zaregistrován ve Sparku, vytvoří se 2 vlákna.
   - první data posílá
   - druhé monitoruje
7. Na konci výpočtu pošle proces zpět speciální značku indikující konec.
8. Spark počká na všechny procesy a všechny výsledky pošle uživateli.

4.5.1 RubyRDD

Prostředník mezi RDD ve Sparku a v Ruby. Jeho jediným úkolem je zajistit výpočet a vrátit zpět výsledek nebo zprávu o chybě. Hlavní parametry při vytvoření objektu jsou:

- Java RDD, která obsahuje data (kolekci).
- Serializovanou třídu `Command`.
- Kde jsou uloženy spustitelné soubory určené pro výpočet.

Výpočet
Ze všeho nejdříve je vytvořen výpočetní proces skrze Master proces. Propojení všech procesů je zajištěno pomocí TCP soketů a ve Sparku se o ně stará objekt `RubyWorker`.

Jakmile se nový proces ohlásí, jsou vytvořena 2 vlákna. Musíme mít na paměti, že daná kolekce může být rozdělena na několik částí. Každá tato část má svůj proces a svoje vlákna. Vlákna jsou důležité pro rychlost. Pokud budeme mít velkou kolekci, může nastat situace, kdy Spark stále posílá data, ale proces již má část dat k odeslání.

První vlákno `WriterThread` postupně posílá data. Data mají formát `|velikost bloku|blok|`. Na konci se odešle speciální znak značící konec. Tento znak je reprezentován jako `velikost bloku = 0`.

Druhé vlákno `MonitorThread` má na starosti hlídání současného výpočtu a případě problému oznámit svému procesu, aby se ukončil. Pokud máme kolekci například rozdělenou na 2 části a v jedné se stane chyba, je žádoucí aby ukončit celý výpočet.

Jakmile je výpočet z nějaké části hotov, Spark je připraven ho přijmout. O to stará třída `StreamReader`, který přijímá 2 druhy zpráv.

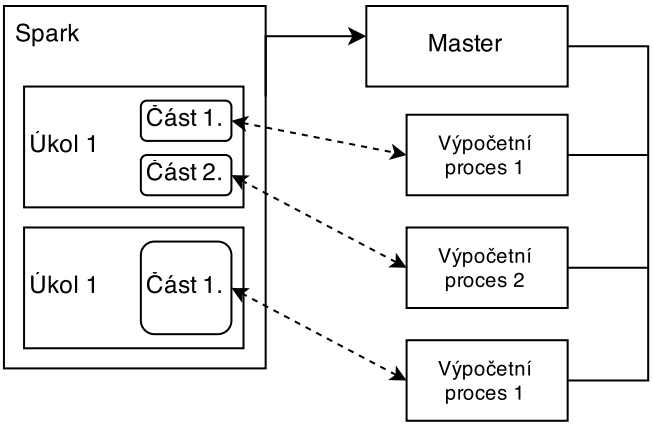
1. Výsledky, ve stejném formátu v jakém data posílal (`|velikost bloku|blok|`).
2. Chyby. Pokud se během procesu vyskytne chyba, proces to oznámí zpět. Navíc přiloží detailní výpis co se stalo.

4.5.2 Master

Je proces, který se stará o všechny výpočetní procesy.

- vytváří procesy
- podle požadavků je ukončuje (posílá `TERM` signál)

Asi se ptáte, proč vytvářet nové procesy z jiného procesu a proč ne rovnou ze Sparku?



Obrázek 4.5: Master proces

První je důsledek operačního systému. Pokud vytvořím proces na UNIXU, tak potomek dostane část adresního prostoru rodiče. To obvykle nepředstavuje problém, ale pokud máme velkou aplikaci jako je Spark, který by navíc vytvářel velké množství dalších procesů, tak se může lehce stát, že dojde paměť. Pokud ale potomky vytvářím z rodiče, který má velice nízké paměťové nároky, tento problém nenastane.

Druhý problém je důsledek implementace Javy, ale také souvisí s prvním. Java totiž nepodporuje operaci fork. Fork je operace vytváření nového procesu, kdy se místo náročného kompletního sestavení prostě zkopíruje paměť toho současného. Ale i kdyby Java fork podporovala, tak máme obdobný problém jako ten minulý. Obdobný, protože nyní se zkopíruje celá paměť. Pokud je ale Master malý a zároveň Ruby proces, tak se oba problémy eliminují.

4.5.3 Výpočetní proces

Je implementací velice jednoduchý. Výpočet probíhá pouze v připravených příkazech, které definuje uživatel. Jeho životní cyklus:

1. Vytvoření socketu do Sparku. Zpět je poslán PID.
2. Výpočet
   a) přečteno číslo části kolekce (důležité pro některé výpočty)
   b) získaná cesta kořenového adresáře (zde mohou být další soubory)
   c) přečtení Broadcastu
   d) přečtení příkazů
   e) přečtení kolekce

f) výpočet

g) odeslání výsledku

3. Došlo k chybě?

   ano: zpět je odeslána chyba

   ne: poslána informace o ukončení

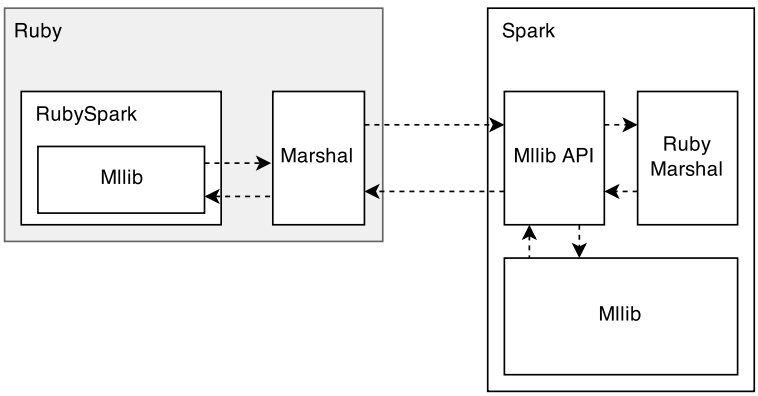
4. Ukončení procesu.

4.6 Mllib

Nyní již víme, že definice i výpočet probíhají v Ruby. Spark je vlastně jen prostředník, který spouští procesy a předává data. V knihovně Mllib je ale jiný přístup. V současné době je tato knihovna velice rozsáhlá a přepsat jí do Ruby by trvalo velice dlouho.

Proč ale kopírovat něco, co je velmi dobře napsáno a optimalizováno pro mnoho platforem? Lepší by bylo vytvořit data a výpočty v Ruby a nechat výpočet na Sparku. A přesně tak je tento nástroj implementován. Ruby je navíc interpretovací programovací jazyk, takže rychlostí se nemůže s kompilovaným jazykem rovnat.

Tento přístup má jediný problém a to *„jak serializovat data?"*. Prošel jsem mnoho knihoven, ale žádná nebyla jak pro Ruby, tak pro Javu. Pokud ano, tak nebyly kompatibilní. Z tohoto důvodu jsem se rozhodl použít Marshal, který má podporu v jádře Ruby. Na straně Sparku jsem ho však musel implementovat.



Obrázek 4.6: RubySpark Mllib

Kapitola 5

Příklady použití

Spark i RubySpark jsou popsány a nyní ukázky použití.

5.1 Instalace

RubySpark je distribuovaný jako Ruby gem a volně dostupný přes rubygems.org. Instalace gemu je velice jednoduchá a stačí spustit:

```
gem install ruby-spark
```

V tuto chvíli máme sice instalovaný gem, ale ne Spark. Ten se musí doinstalovat ručně. Automaticky není tato možnost dostupná, protože balíček Sparku má okolo 120MB. Uživatel navíc může použít svůj vlastní zkompilovaný Spark. RubySpark je po instalaci dostupný jako `ruby-spark`. Pro základní sestavení stačí:

```
ruby-spark build
```

Pokud chceme sestavení modifikovat, existuje 5 přepínačů:

`--hadoop-version` konkrétní verze hadoop

`--spark-home` kde je či má být domovský adresář Sparku

`--spark-version` verze Sparku

`--scala-version` verze Scaly, pod kterou je Spark sestaven

`--only-ext` budou zkompilované pouze rozšíření nutné pro Ruby

Spark i RubySpark jsou instalovány a připraveny k použití.

5.2 Spuštění

RubySpark lze spustit v nějakém Ruby projektu či jako interaktivní konzoli. Jakmile je Spark nastartován, nelze měnit jeho konfiguraci.

5.2.1 Konfigurace

Využívá stejný mechanismus jako konfigurace ve Sparku. Proto nové hodnoty definované pro RubySpark jsou vidět i ve webovém rozhraní. Konfigurovat lze přímo v Ruby pomocí:

```
Spark.config do
  set KLÍČ, HODNOTA
  set 'spark.ruby.serializer', 'oj'
end

Spark.config.set(KLÍČ, HODNOTA)
Spark.config.set(set 'spark.ruby.serializer', 'oj')
```

nebo pomocí proměnných prostředí

```
export SPARK_RUBY_SERIALIZER='oj'
```

Klíč	Popis
`spark.ruby .worker.type`	**process:** výpočty jsou prováděny v procesech **thread:** výpočty pomocí vláken (experimentální) *výchozí: process*
`spark.ruby .worker.arguments`	Extra argumenty použité při spuštění procesů.
`spark.ruby .parallelize_strategy`	Data vložená pro paralelizaci mohou být změněna z důvodu serializace či komprese. Pro takové případy lze rozhodnout, co se s původními daty stane. **inplace:** data se změní **deep_copy:** data jsou nejdříve zduplikována, aby nedošlo k jejich přepsání *výchozí: inplace*
`spark.ruby .serializer`	Výchozí serializátor. *výchozí: marshal*
`spark.ruby .batch_size`	Kolik položek bude serializováno do jedné. *výchozí: 1024*

5.2.2 Spuštění v konzoli

Pomocí

```
ruby-spark shell
```

se spustí kompletně připravená konzole a lze přejít rovnou na nahrání dat. Pokud ale chceme RubySpark předtím nakonfigurovat, lze konzoli spustit v nenastartovaném módu.

```
ruby-spark shell --no-start
```

V tomto prostředí lze nakonfigurovat Spark a následně jej nastartovat a získat context.

```
# Start
Spark.start

# Context
Spark.context
```

5.2.3 Spuštění v Ruby projektu

Je stejně jednoduché jako v případě použití konzole bez nastartování.

```
require 'ruby-spark'

# Konfigurace
Spark.config.set(..., ...)

# Start
Spark.start

# Context
Spark.context
```

5.3 Nahrání dat

Data můžeme vkládat přímo jako struktury Ruby pomocí `.parallelize`, jako soubor přes `.text_file` nebo všechny soubory ve složce s `.whole_text_files`. Soubory mohou být uloženy na lokálním souborovém systému, HDFS nebo v nějakém podporovaném Hadoop protokolu. Všechny soubory musí být kódované v UTF-8. Ostatní kódovaní sice bude fungovat také, ale nemusíme dostat správný výsledek.

Během nahrání dat lze nastavit následující parametry:

`workers_num` na kolik části budou data rozdělena (odpovídá počtu výpočetních procesů)

`serializer` jméno serializátoru, který bude data serializovat

`batch_size` kolik položek bude v jedné serializované položce

Ukázka:

```
spark.parallelize(
   DATA, workers_num, serializer: ..., batch_size: ...)

spark.text_file(
   SOUBOR, workers_num, serializer: ..., batch_size: ...)

spark.whole_text_files(
   ADRESÁŘ, workers_num, serializer: ..., batch_size: ...)
```

5.4 Definice výpočtu a jeho aktivace

Problém serializace funkcí byl popsán v kapitole 4.3. V této kapitole budou pouze funkční ukázky a přístupy.

Funkce jako Proc
Klasické definování anonymní funkce a její vložení jako parameter výpočetní metody.

```
func = lambda{|x| x*2}
spark.map(func)
```

Funkce jako text
Pokud bude problém s prvním způsobem, tak ho lze vyřešit vložení funkce rovnou jako text.

```
func = "lambda{|x| x*2}"
spark.map(func)
```

Zkrácený zápis
Pokud chceme, aby nad každou položkou kolekce byla volána jedna konkrétní metoda, tak lze použit zkrácený zápis. Například chceme číslo převést na jeho textový ekvivalent. V Ruby stačí nad číslem zavolat `.to_s`.

```
spark.map(:to_s)
```

Metody
Kromě anonymních funkcí lze vkládat také ty pojmenované. K tomu musíme použít funkci z jádra `method()`, která vrátí referenci na danou metodu. Tento postup má ale podobný problém jako u anonymních funkcí.

```
def nasobeni(x)
  x*2
end

spark.map(method(:nasobeni))
```

Jakmile jsou všechny požadované operace nadefinovány, lze spustit výpočet. Některé operace spustí výpočet samy (například `.count`), ale pro ostatní je nutné volat `.collect`.

5.5 Vázané objekty

Tímto způsobem lze částečně vynahradit problém serializace uzávěrů. Pro malé zopakování příklad. Proměnné nebudou součástí (v příkladu `nums`) serializovaných dat a tudíž se ve výpočtu stane chyba, která ho ukončí.

```
nums = 3
func = lambda{|x| nums*x*2}

spark.map(func)
```

Pro takové případy jsem implementoval metodu `.bind()`, které připojí zvolené proměnné. Jako parametr je vložen Hash, kde klíč představuje jméno proměnné a hodnota jeho hodnotu ve výpočetním procesu.

```
nums = 3
func = lambda{|x| nums*x*2}

spark.map(func).bind(nums: nums)
```

5.6 Sdílené proměnné

Stejně jako Spark i RubySpark podporuje sdílené proměnné Broadcast a Accumulator. Kvůli serializačním problémům se však musí i zde použít jiné řešení.

5.6.1 Broadcast

Broadcast je i zde pouze ke čtení a tudíž jako parametr vyžaduje pouze danou hodnotu.

```
# Definice
broadcast = context.broadcast(3)

# Funkce
func = lambda{|x| broadcast.value * x * 2}

# Mapování
spark.map(func).bind(broadcast: broadcast)
```

5.6.2 Akumulátor

Je proměnná se dvěma módy.

1. V řídícím programu ji lze jakkoliv upravit.
2. Ve výpočetním procesu lze pouze inkrementovat nulovou hodnotu.

`accumulator = context.accumulator(value, accum_param, zero_value)`

`value` Počáteční hodnota.

`accum_param` Jak se budou inkrementovat hodnoty ve výpočetním procesu. Může být vložen buď zástupný symbol (+,-,*,/) nebo funkce se dvěmi parametry.

`zero_value` Počáteční hodnota ve výpočetním procesu.

Ukázka dvou variant:

```
# Sčítací
accum1 = context.accumulator(0, :+, 0)

# Hledání maxima
max = lambda{|max, val| val > max ? val : max}
accum2 = context.accumulator(0, max, 0)

# Funkce
func = lambda{|x|
  accum1.add(x)
  accum2.add(x)
  x * 2
}

# Mapování
spark.map(func).bind(accum1: accum1, accum2: accum2)
```

Kapitola 6

Vyhodnocení

Kapitola popisuje výsledky testování výpočtu na Sparku mezi:

- Ruby za použití Marshal serializatoru
- Ruby za použití Oj serializatoru
- Python
- Scala

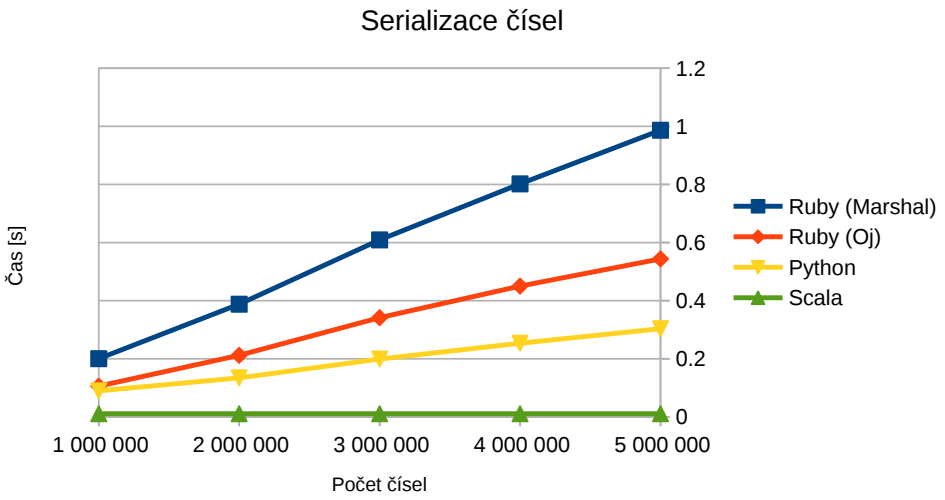
Původně jsem chtěl do výsledku zahrnout také jazyk R, nicméně ten nedokázal zpracovávat větší množství dat. Při množství, které zvládal, zase ostatní platformy vykazovaly tak malé časy, že to v podstatě nemělo žádnou vypovídající hodnotu.

Součásti testu bylo pouze Ruby s C interpreterem (MRI). JRuby, které podporuje všechny funkce a možnosti RubySparku, se stále nachází ve vývojové verzi a jeho časy byly minimálně 10x vyšší.

Vítěz testu je ve všech bodech Scala. Ostatní jazyky se k ní téměř nepřiblížily. Na druhém místě byl Python a bohužel poslední Ruby. Ačkoliv tento jazyk není vyvíjen pro složité a dlouho trvající výpočty, tak nakonec ani nezklamal. To také dokazuje výsledek ve výpočtu čísla Pi, kdy dosáhl lepších výsledku než Python.

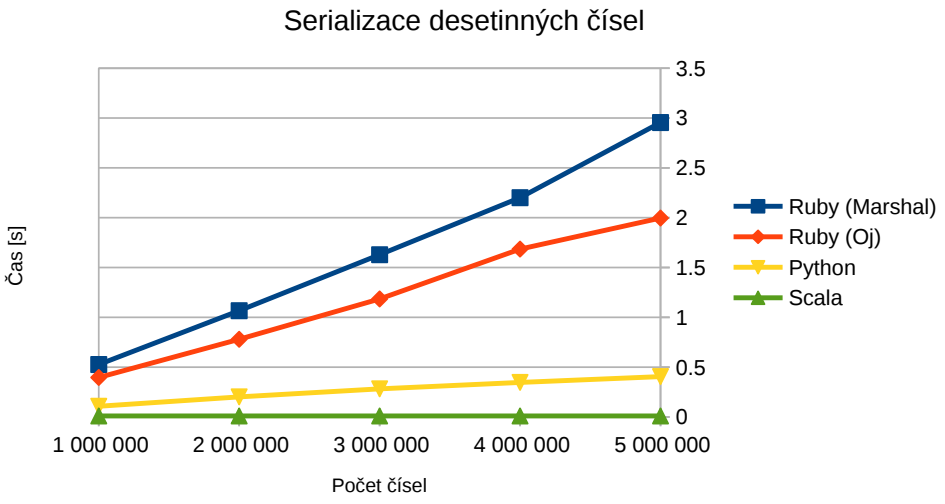
6.1 Serializace

První test se týkal načtení obyčejných čísel. Zde je velmi dobře vidět, jak závisí čas na zvoleném serializátoru. Nicméně rozdíl se pohybuje kolem půl sekundy, takže při dlouhých výpočtech lze tento rozdíl zanedbat.



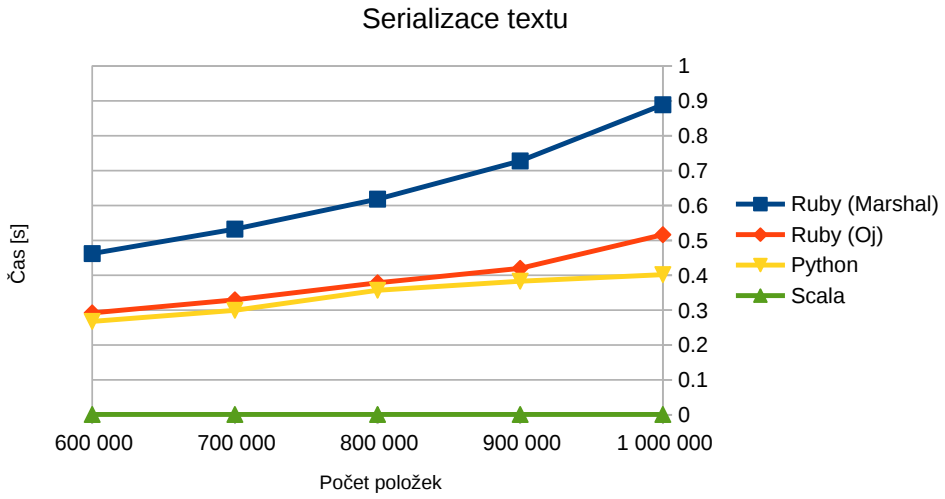
Obrázek 6.1: Serializace čísel

Podobný prvnímu testu. Místo jednoduchých čísel se ale použily desetinná čísla.



Obrázek 6.2: Serializace čísel

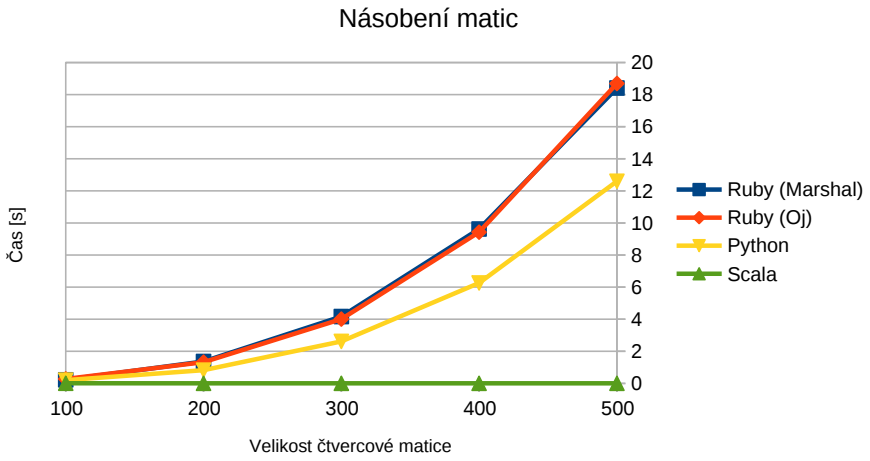
Poslední serializační test obsahoval text. Každá položka představuje větu o velikosti 10 slov.



Obrázek 6.3: Serializace textu

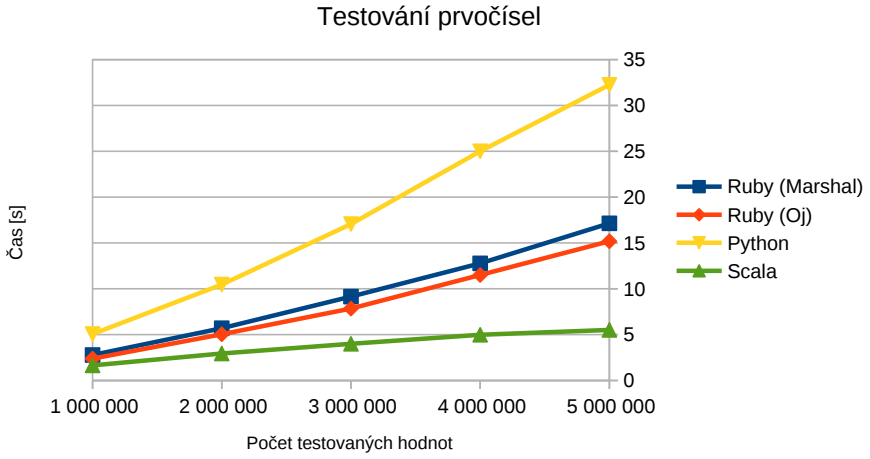
6.2 Počítání

Test měl za úkol násobit matice s využitím základní struktury pole v každém jazyce. Není zde použita žádná optimalizace, abych žádný jazyk nezvýhodnil. Matice má čtvercový tvar a je násobena sama sebou. Nyní lze vidět, že pro výpočet už nezáleží na zvoleném serializačním nástroji.



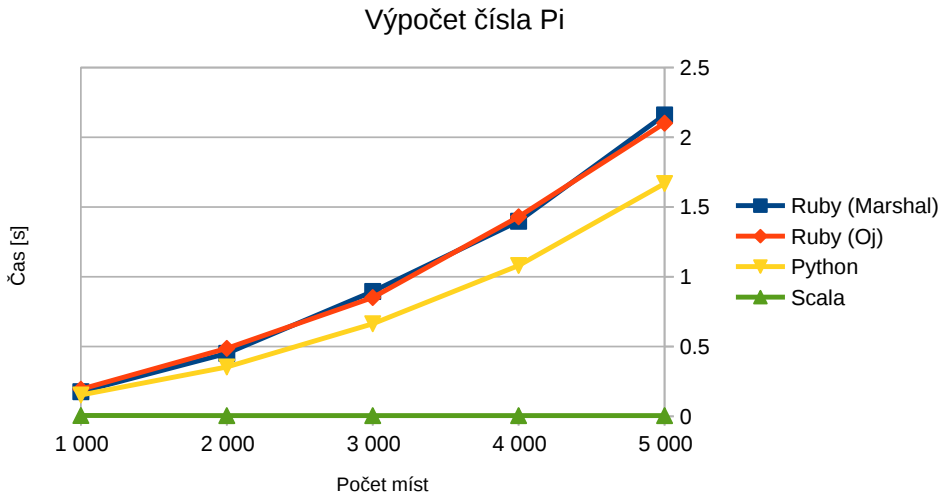
Obrázek 6.4: Násobení matic

Jediný test, kde Ruby předčil Python. Jedná se o jednoduché testování, zda je číslo prvočíslo. Testuje se soudělnost na všechny menší lichá čísla.



Obrázek 6.5: Testování prvočísel

Poslední test se týkal výpočtu čísla pi na určitý počet desetinných míst.



Obrázek 6.6: Výpočet čísla Pi

Závěr

Hlavním cílem této diplomové práce bylo přidání podpory jazyka Ruby do projektu Apache Spark. Tento cíl byl splněn a byla vytvořena Ruby knihovna, která poskytuje stejnou funkcionalitu jako jádro Sparku. Ačkoliv to nebylo součástí zadání, byla navíc přidána podpora Mllib knihovny, která obsahuje metody regrese, klasifikace i shlukové analýzy.

Nicméně zde stále zůstávají věci, které chybí (oproti Sparku) či by mohly být vylepšeny. Chybí například podpora streamování nebo SQL. Vylepšena by mohla být komprese serializovaných dat, získání více informací o proběhlém výpočtu (použití pamětí, čas trvání) a podpora dalších serializačních nástrojů. Také bych si přál vylepšit serializaci funkcí, která je sice funkční, nicméně uživatel si musí být vědom možných problémů.

Díky této diplomové práci jsem měl možnost seznámit se s projektem, který je v dnešní době velice populární a zaujímá důležitou pozici při zpracování dat v mnoha firmách a nástrojích. V budoucnosti bych rád tuto práci dále rozšiřoval a staral se o její vývoj.

Literatura

[1] van't Spijker, A.: *The New Oil: Using Innovative Business Models to turn Data Into Profit*. 2014, ISBN 9781634620567. Dostupné z: https://books.google.cz/books?id=zY3XBgAAQBAJ

[2] Hastie, T.; Tibshirani, R.; Friedman, J.: *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. Springer Series in Statistics, Springer, 2009, ISBN 9780387848587. Dostupné z: https://books.google.cz/books?id=tVIjmNS3Ob8C

[3] Karau, H.; Konwinski, A.; Wendell, P.; aj.: *Learning Spark: Lightning-Fast Big Data Analysis*. O'Reilly Media, 2015, ISBN 9781449359065. Dostupné z: https://books.google.cz/books?id=2eptBgAAQBAJ

[4] Venner, J.; Wadkar, S.; Siddalingaiah, M.: *Pro Apache Hadoop*. Apress, 2014, ISBN 9781430248644. Dostupné z: https://books.google.cz/books?id=uL59BAAAQBAJ

[5] Ryza, S.; Laserson, U.; Owen, S.; aj.: *Advanced Analytics with Spark: Patterns for Learning from Data at Scale*. O'Reilly Media, 2015, ISBN 9781491912713. Dostupné z: https://books.google.cz/books?id=M0_GBwAAQBAJ

[6] Pentreath, N.: *Machine Learning with Spark*. 2015, ISBN 9781783288526. Dostupné z: https://books.google.cz/books?id=syPHBgAAQBAJ

[7] Srinivasa, K.; Muppalla, A.: *Guide to High Performance Distributed Computing: Case Studies with Hadoop, Scalding and Spark*. Computer Communications and Networks, Springer International Publishing, 2015, ISBN 9783319134970. Dostupné z: https://books.google.cz/books?id=wWCYBgAAQBAJ

[8] Sankar, K.; Karau, H.: *Fast Data Processing with Spark - Second Edition*. 2015, ISBN 9781784399078. Dostupné z: https://books.google.cz/books?id=YTe9BwAAQBAJ

[9] Anthony, R.: *Systems Programming: Designing and Developing Distributed Applications*. Elsevier Science, 2015, ISBN 9780128008171. Dostupné z: https://books.google.cz/books?id=BuCcBAAAQBAJ

[10] Fasale, A.; Kumar, N.: *YARN Essentials*. 2015, ISBN 9781784397722. Dostupné z: https://books.google.cz/books?id=DrfNBgAAQBAJ

[11] Murthy, A.; Vavilapalli, V.; Eadline, D.; aj.: *Apache Hadoop YARN: Moving beyond MapReduce and Batch Processing with Apache Hadoop 2*. Addison-Wesley Data & Analytics Series, Pearson Education, 2014, ISBN 9780133441918. Dostupné z: https://books.google.cz/books?id=heoXAwAAQBAJ

[12] Fulton, H.; Arko, A.: *The Ruby Way: Solutions and Techniques in Ruby Programming*. Addison-Wesley Professional Ruby Series, Pearson Education, 2015, ISBN 9780132480376. Dostupné z: https://books.google.cz/books?id=uT6eBgAAQBAJ

[13] Perrotta, P.: *Metaprogramming Ruby 2: Program Like the Ruby Pros*. Facets of Ruby, Pragmatic Bookshelf, 2014, ISBN 9781941222126. Dostupné z: https://books.google.cz/books?id=V0iToAEACAAJ

[14] Carlson, L.; Richardson, L.: *Ruby Cookbook*. O'Reilly Media, 2015, ISBN 9781449373672. Dostupné z: https://books.google.cz/books?id=xBmkBwAAQBAJ

[15] Shaughnessy, P.: *Ruby Under a Microscope: Learning Ruby Internals Through Experiment*. No Starch Press, 2013, ISBN 9781593275273. Dostupné z: https://books.google.cz/books?id=P7AdAgAAQBAJ

[16] McDonough, P.: Parallel programming with Spark. 2013. Dostupné z: http://spark-summit.org/talk/parallel-programming-with-spark

[17] Hindman, B.; Konwinski, A.; Zaharia, M.; aj.: Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. University of California, Berkeley, 2010. Dostupné z: http://mesos.berkeley.edu/mesos_tech_report.pdf

[18] Dokumentace Apache Spark. Dostupné z: http://spark.apache.org/docs/latest/index.html

[19] Dokumentace Apache Mesos. Dostupné z: http://mesos.apache.org/documentation/latest

[20] Support Vector Machines in Python. Dostupné z: http://www.mblondel.org/journal/2010/09/19/support-vector-machines-in-python

[21] Murthy, A.: Apache Hadoop YARN – Concepts and Applications. 2012. Dostupné z: http://hortonworks.com/blog/apache-hadoop-yarn-concepts-and-applications

[22] Shaughnessy, P.: Visualizing Garbage Collection in Ruby and Python. 2013. Dostupné z: http://patshaughnessy.net/2013/10/24/visualizing-garbage-collection-in-ruby-and-python

Příloha A

Seznam použitých zkratek

RDD Resilient Distributed Dataset

MRI Matz's Ruby Interpreter

GC Garbage Collector

JVM Java Virtual Machine

JNI Java Native Interface

YAML YAML Ain't Markup Language

XML Extensible Markup Language

JSON JavaScript Object Notation

TCP Transmission Control Protocol

PID Process IDentifier

HDFS Hadoop Distributed File System

UCS Universal Character Set

UTF UCS Transformation Format

SQL Structured Query Language

Příloha **B**

Obsah přiloženého CD

- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
  - `ruby-spark.gem` ..... RubySpark knihovna
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis.ps` ..... text práce ve formátu PS
- `readme.txt` ..... stručný popis obsahu CD