MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Využitelnost systému MapReduce pro aplikace v NLP

DIPLOMOVÁ PRÁCE

Bc. Pavel Hančar

Brno, Podzim 2013

Prohlášení

Prohlašuji, že tato diplomová práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje, prameny a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

Vedoucí práce: Mgr. Pavel Rychlý, Ph.D.

Poděkování

Děkuji svému vedoucímu Pavlovi Rychlému, Miloši Jakubíčkovi a kolektivu CZPJ za cenná nasměrování při práci. Za velkou ochotu a trpělivost děkuji správcům z CVT, jejichž zkušenosti se mi velmi hodili. Děkuji svým kamarádům a rodině za podporu.

MapReduce, Hadoop, HDFS, DDFS, Disco,

Obsah

Kapitola 1

Úvod

S rozmachem elektronicky uložených dat roste i potřeba velké objemy dat zpracovávat. Velmi dobře si s tímto úkolem poradila firma Google. V roce 2004 představila programové schéma MapReduce, které umožňuje velmi jednoduché programování paralelního zpracování velkých objemů dat.

Zpracováním velkých objemů textových dat se zabývá i korpusová lingvistika. Korpusem je pro ni rozsáhlý soubor textů, upravených tak, aby byly cenné z hlediska lingvistických výzkumů. Korpusy pak často slouží k realizaci úloh statistických přístupů k NLP (Natural Language Processing).

Sběr a zpracování korpusových dat je jednou z hlavních činností Centra zpracování přirozeného jazyka (CZPJ) na Fakultě informatiky Masarykovy Univerzity(FI MUNI). Proto přišla na řadu otázka, zda by použití MapReduce bylo výhodné i v podmínkách CZPJ, případně jakou implementaci tohoto schématu zvolit.

Kapitola 2 se věnuje obecnému popisu schématu MapReduce. Kapitola 3 obsahuje popis jeho rozšířené implementace Hadoop, zatímco kapitola 4 představuje méně rozšířenou odlehčenou implementaci jménem Disco. Instalaci obou těchto implementací v rámci CZPJ a FI MUNI představuje kapitola 5. Kapitola 6 představuje úlohy, které tento software může v rámci CZPJ řešit, představuje náhradní jednodušší řešení založené na jazyce příkazové řádky systému UNIX a srovnává uvedené přístupy na naměřených časech zpracování vybraných úloh. Závěr shrnuje výsledky a přílohy obsahují tabulky naměřených hodnot a zdrojové kódy programů představených v rámci práce.

Kapitola 2

MapReduce

Tato kapitola se pokouší vysvětlit základní koncepci programového schématu MapReduce. Více podrobností naleznete v článku [4].

Úlohy zpracování textových dat (např. získávání invertovaných indexů, přehledů stažených stránek, grafových struktur webových stránek atp.) jsou vesměs koncepčně jednoduché. Aby ovšem výpočet skončil v rozumném čase, je nutné jej rozdělit mezi velké množství strojů. Distribuování výpočtu znepřehledňuje původně jednoduché řešení velkým množstvím složitého kódu. Z tohoto důvodu navrhla firma Google abstrakci, která umožňuje skrýt paralelizaci, rozdělení dat a vyvážení zátěže do knihovny. Touto abstrakcí je programové schéma MapReduce.

Název MapReduce je inspirovaný funkcionálními jazyky. **Map** v názvu značí namapování unární funkce na seznam[1] a **Reduce** znamená redukování seznamu pomocí binární funkce[2]. V kontextu MapReduce se zmíněným seznamem rozumí textová data rozdělená na menší segmenty (např. řádky).

Proč právě tyto dvě funkce? Řekněme, že chceme problém paralelního zpracování velkých textových dat řešit tím, že jednoduše *rozdělíme data mezi více počítačů a spustíme zpracování paralelně*. To lze použít třeba pro triviální úlohy, jako je změna kódování textu, nebo i některé komplikované (morfologická či syntaktická analýza). Problém ale nastane např. s úlohou spočtení četností slov. Při rozdělení dat mezi počítače by byly výstupem pouze četnosti slov na jednotlivých strojích.

Obecně totiž existují dva základní typy úloh, které můžeme ve shodě se schématem pojmenovat **typ map** a **typ reduce**:

Typ map znamená, že segment výstupu[3] je závislý pouze na jednom segmentu vstupu.

Typ reduce znamená, že segment výstupu je závislý na více segmentech vstupu rozmístěných v celé jeho šíři.

Složitější úlohy typu reduce se mohou skládat z celé posloupnosti podúloh typu map a reduce. Pro většinu běžných úloh typu reduce ovšem stačí jedna podúloha typu map, která upraví vstupní data (typicky je rozdělí na segmenty nebo ohodnotí) a jedna podúloha typu reduce. Proto schéma MapReduce.

2.1 Příklad – četnost slov v textu

Práce [4] zavádí funkce *map* a *reduce* na dvojicích $\langle klic, hodnota \rangle$. Spočtení četností slov potom vypadá takto:

---

1. Přesněji aplikaci unární funkce na prvky seznamu – v Haskellu se jmenuje map, v Lispu $mapcar$
2. Přesněji aplikace binární funkce nejprve na parametr a poslední prvek, potom na mezivýsledek a předchozí prvek – v Haskellu je to $foldr$, v Common Lisp je to $reduce$.
3. např. slovo nebo věta

1. Vstupní text se rozdělí na seznam dvojic $\langle klic, hodnota \rangle$ ve tvaru: $\langle pozice\ radku\ v\ souboru, radek \rangle$ [4]

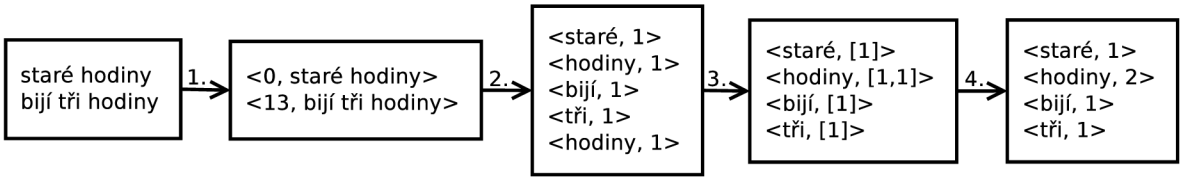
2. Funkce map namapuje jedničku (tj. počáteční četnost) ke každému slovu:

$$map : \langle pozice\ radku\ v\ souboru,\ radek \rangle \mapsto [\langle 1.\ slovo\ radku, 1 \rangle, \langle 2.\ slovo\ radku, 1 \rangle, \ldots]$$

3. Dvojice $\langle slovo, cislo \rangle$ se seskupí podle slov.

4. Funkce $reduce$ dostane na vstup jedno slovo a seznam jeho dílčích četností, které sečte:

$$reduce : \langle slovo, [cetnost_1, cetnost_2, \ldots] \rangle \mapsto \langle slovo, [cetnost_1 + cetnost_2 + \ldots] \rangle$$



Výhodou schématu je jeho jednoduchost. Pro většinu úloh uživateli stačí specifikovat právě jenom funkce map a $reduce$, příp. jednu z nich. V příkladu jsou obě, jako body 2. a 4.

Obecně mají map a $reduce$ typy:

$$map : (k_1, v_1) \to seznam(k_2, v_2)$$

$$reduce : (k_2, seznam(v_2)) \to (k_2, seznam(v_2))$$

Kde k_i, v_j jsou typy klíčů a hodnot a (k_i, v_j) je množina všech uspořádaných dvojic na těchto typech.

2.2 Realizace výpočtu v clusteru

1. Data jsou rozdělena na M dílů a distribuována po clusteru. Na každém počítači je spuštěna kopie MapReduce.

2. Jedna z kopií MapReduce je speciální (tzv. řídící – ostatní jsou podřízení). Řídící postupně pověřuje nevyužité podřízené provedením funkcí map na M blocích vstupních dat a funkcí $reduce$ na R blocích.

3. Podřízení zpracují vstupní data, provedou na nich funkce map a uloží si mezivýsledky ve tvaru $\langle klic, hodnota \rangle$ do vyrovnávací paměti.

4. Z vyrovnávací paměti jsou mezivýsledky periodicky zapisovány na lokální disk do R oblastí jejichž umístění jsou předána řídícímu, který je přepošle na podřízené pověřené funkcemi $reduce$.

5. Každý podřízený pověřený funkcemi $reduce$ načte data z podřízených s mezivýsledky, seskupí stejné klíče (např. použitím třídění) a provede pro ně funkce $reduce$.

4. Toto je implicitní vstupní formát v implementaci jménem Hadoop. V tomto příkladu by stačil jakýkoli klíč neboť se ignoruje. Hodnota zase může být i větší úsek textu než řádek, ale musí se vejít do paměti počítače a neměl by dělit slova.

Po provedení tohoto postupu je v clusteru k dispozici R souborů s výsledky, které lze zobrazit nebo využít pro další MapReduce úlohy.

Ctěným principem je v MapReduce lokalita dat, protože nadměrné kopírování rozsáhlých dat po clusteru může síť zbytečně zahltit a navíc představuje pro systém nadbytečnou režii. Proto úlohy map pracují výhradně lokálně. To ovšem neplatí pro úlohy $reduce$. Ty musí získat všechny mezivýsledky s daným klíčem, jenž jsou po provedení map rozprostřeny po clusteru. Z toho důvodu definuje MapReduce ještě funkci $combiner$, která má stejný typ jako $reduce$, ovšem může se lišit a je prováděna lokálně. Často se definuje $combiner = reduce$. To lze ovšem jen u komutativních asociativních funkcí. Nelze to například pro aritmetický průměr, kde je ovšem možné ve funkci $combiner$ spočítat součet a počet čísel, a ty pak předat funkci $reduce$, která počítá samotný aritmetický průměr (tedy $combiner \neq reduce$).

2.3 Volby uživatele

Funkce map a $reduce$ ale nejsou tím jediným, co uživatel může zadat. Zde je výčet těch hlavních možností:

- Uživatel může definovat způsob rozdělení vstupních dat do dvojic $\langle klic, hodnota \rangle$.
- Uživatel definuje funkci $map : \langle klic_1, hodnota_1 \rangle \rightarrow \langle klic_2, hodnota_2 \rangle$.
- Uživatel může definovat funkci $combiner$.
- Uživatel může definovat funkci $reduce$.
- Uživatel může upravit formát výstupu.
- Uživatel může specifikovat počet funkcí $reduce$, které se provedou.

Kapitola 3

Hadoop

Tato kapitola se pokouší shrnout vše důležité, co potřebuje uživatel znát, pokud chce používat framework Hadoop a zároveň rozumět jeho základům. Více informací je v knize [6].

Hadoop je volně dostupná implementace MapReduce. Vytvořil ji Doug Cutting, tvůrce Apache Lucene, široce používané knihovny pro vyhledávání v textu. Hadoop má původ v Apache Nutch, volně dostupném webovém vyhledávači, který je součástí projektu Lucene.

Vznik jména vysvětluje Doug Cutting takto: *To jméno dalo mé dítě žlutému plyšovému slonovi. Krátké, poměrně snadné napsat a vyslovit, bez významu a nepoužívané jinde: to jsou má kritéria pro jména. Děti jsou dobré ve vytváření takových.*

Aby mohli tvůrci Nutch spravovat miliardy webových stánek, inspirovali se distribuovaným souborovým systémem GFS, který v roce 2003 představila firma Google. V roce 2004 tak vytvořili Nutch Distributed File System (NDFS – později přejmenovaný na HDFS). V roce 2004 představil Google MapReduce a už v roce 2005 měl Nutch svoji implementaci, která se v roce 2006 osamostatnila a vytvořila v rámci Lucene nový projekt jménem Hadoop. V roce 2006 také vstoupil Doug Cutting do Yahoo!, jež se pak brzy rozhodlo adaptovat Hadoop do svého WebMap (části vyhledávače, která buduje graf známého webu). V roce 2008 Hadoop vyhrál soutěž v třídění jednoho terabajtu dat. V té době ho kromě Yahoo! používaly již společnosti jako Facebook, Last.fm nebo New York Times.

3.1 Verze systému Hadoop

Základní distribuce je Apache Hadoop – v době psaní práce byla v těchto verzích:

1.2.X současná stabilní verze

2.2.X současná stabilní verze 2

0.23.X podobná 2.X.X ale chybí NN HA (Highly Available NameNode)

0.22.X neobsahuje bezpečnost

0.20.X stará verze

Další známé distribuce nabízí firmy Hortonworks a Cloudera. V této práci je použita distribuce firmy Cloudera ve verzi CDH3 update 1, která vychází z verze 0.20.2 (tedy verze již starší, zato stabilní). Větev CDH3 se v době psaní práce sice dostala až na update 6, ale kvůli problémům s lokální instalací práce zůstala u verze CDH3u1.

Hadoop verze 2 nahrazuje schéma MapReduce jeho zobecněním jménem Apache Tez. To umožňuje snadno zadávat instance vykonávající map a $reduce$ seřazené v orientovaném acyklickém grafu a zároveň zajišťuje efektivnější provedení těchto složitých úloh. je ovšem nad rámec této práce. Hadoop verze 2 má rovněž jinou architekturu, tzv. YARN.

3.2 Ekosystém Hadoopu

Termín Hadoop je používán pro implementaci MapReduce nad distribuovaným souborovým systémem HDFS, ale někdy i pro celou rodinu projektů pro distribuované zpracování rozsáhlých dat. Následuje stručný popis některých:

Common Množina komponent a rozhraní pro distribuované souborové systémy a vstup/-výstup obecně (serializaci, Java RPC – neboli vzdálené volání procedur, trvalé datové struktury).

Avro Serializační systém pro efektivní trvalé datové úložiště s RPC nezávislém na jazyku.

MapReduce Model pro distribuované zpracování dat a jeho implementace pro rozsáhlé clustery běžných počítačů.

HDFS Distribuovaný souborový systém používaný na rozsáhlých clusterech běžných počítačů. Je hlavním představitelem, ale nikoli jediným (alternativou je např. Amazon S3).

Pig Jazyk datových toků a jeho běhové prostředí pro analýzu rozsáhlých kolekcí dat.

Hive Distribuovaný datový sklad. Hive řídí data uložená v HDFS a poskytuje dotazovací jazyk založený na SQL (který je za běhu překládán na úlohy MapReduce).

HBase Distribuovaná sloupcově orientovaná databáze. HBase pro své úložiště používá HDFS a podporuje jak dávkově orientované výpočty pomocí MapReduce, tak dotazy na menší datové objekty.

ZooKeeper Centrální služba pro správu informací o konfiguraci, distribuovanou synchronizaci a poskytování skupinových služeb. (přeloženo z [2])

Sqoop Nástroj pro efektivní přesun dat mezi relačními databázemi a HDFS.

(Převzato z [6] str. 12, 13, kde je více informací.)

3.3 Hadoop Distributed File System

Základem téměř všech součástí systému Hadoop je distribuovaný souborový systém potřebný k ukládání dat. Tato sekce představuje základní rysy HDFS, tedy souborového systému nejčastěji používaného systémem Hadoop. Více infomací je v článku [3] a v knize [6].

HDFS je distribuovaný souborový systém navržený pro uložení velmi velkých souborů[1] na běžně dostupných počítačích (*commodity hardware*). Je navržen spíše pro dávkové zpracování dat než pro interaktivní použití. Důraz je kladen spíše na vysokou propustnost, než na nízkou dobu odezvy přístupu k datům.

HDFS je napsaný v Javě, a tvoří další vrstvu nad souborovým systémem operačního systému. Poskytuje rozhraní na způsob standardu POSIX, ale některé požadavky tohoto standardu nesplňuje výměnou za lepší propustnost dat. Příkazy známé z UNIXu (např. `ls`, `cp`, `cat`, ...) lze zadat jako argumenty příkazu `hadoop` např. takto:

```
hadoop dfs -ls <path>
```

1. Stovky megabajtů, gigabajty, terabajty případně až petabajty dat.

3.3.1 Architektura a implementace

HDFS je aplikace typu řídící/podřízený (master/slave). Řídící spravuje strom souborů a jejich metadata, podřízený ukládá a zprostředkovává data a pravidelně o nich informuje řídící.

Namenode je označení démona, který implementuje řídící uzel, a počítače, na kterém běží. Řídí přístupy k HDFS a replikaci bloků dat (viz 3.3.2). Dále přijímá od démonů datanode (viz níže) tzv. `Heartbeat` (potvrzení správného fungování datanode) a `Blockreport` (seznam bloků na počítači daného datanode). Udržuje v paměti aktuální verzi stromu souborů a metadat, přičemž změny zaznamenává (v adresáři `${dfs.data.dir}/current`[2] viz 3.5) do souboru *edits*. Stará podoba stromu je (v tomtéž adresáři) v souboru *fsimage*. Strom se aktualizuje při restartu nebo ho aktualizuje secondary namenode (viz níže). Běží zpravidla na jednom samostatném počítači. Pro případ selhání je ovšem dobré, když ukládá zmíněné soubory ještě na jiný stroj přes NFS. Jeho webové rozhraní je na portu 50070.

Datanode je označení démona, který implementuje podřízený uzel, a počítače, na kterém běží. Ukládá data do bloků a komunikuje s namenode. Běží zpravidla na všech počítačích, kde není namenode nebo secondary namenode.

Secondary namenode je označení démona a počítače, na kterém běží. Jméno je však zavádějící, protože tento démon se nechová jako namenode. Jeho úlohou je pravielně aktualizovat soubor *fsimage* podle souboru *edits* na namenode, aby soubor *edits* nerostl do nekonečna (to by mohlo fatálně zpomalit restart namenode). Zmíněná aktualizace probíhá následovně: jednou za hodinu (popř. při dovršení 64 MB souborem *edits*, kontrolovaném každých pět minut) stahuje soubory *fsimage* a *edits* z namenode, provede aktualizaci, nahraje aktuální *fsimage* na namenode a poznamená čas do souboru *fstime*. Běží zpravidla na jednom samostatném počítači. Poskytuje tedy zálohu pro namenode, ale zaostává za ním, takže při naprostém selhání počítače s namenode je ztráta dat téměř jistá a běžný postup je zkopírovat náhradní data z NFS a spustit nový namenode místo secondary namenode. Jeho webové rozhraní je na portu 50090.

3.3.2 Replikace

HDFS ukládá každý soubor jako posloupnost stejně velkých bloků (až na poslední). Pro odolnost vůči chybám jsou bloky replikovány (ukládány ve více kopiích). Faktor replikace (počet kopií) a velikost bloku jsou nastavitelné pro každý soubor. Typicky se ale nespecifikují a použije se hodnota specifikovaná v konfiguraci. Implicitně je faktor replikace 3 a velikost bloku 64 MB.

HDFS optimalizuje umístění replik. Cluster lze rozdělit na stojany (racks) a repliky umísťovat tak, aby se vyvážily na jedné straně spolehlivost (třeba i pro případ výpadku celého stojanu) a na druhé nároky na šířku pásma. Strategií pro umísťování replik je více. Implicitně HDFS umístí první repliku – pokud možno – lokálně (jinak náhodně), druhou repliku na jiný (náhodně vybraný) stojan než první, třetí repliku na stejný stojan jako druhou, ale na jiný (náhodně vybraný) stroj. Případné další repliky se umísťují na náhodný stroj. Systém se ovšem snaží vyhnout umístění příliš mnoha replik na stejný stojan.

2. Kde volba konfigurace `dfs.data.dir` je cesta v souborovém systému OS.

Když systém stanoví umístění replik, vytvoří se tzv. `replica pipeline` (replikační roura). To znamená, že klient uloží pouze jednu repliku na nějaký uzel lokálního stojanu. Ten uloží další repliku atd.

Časem se distribuce bloků může stát nevyváženou. Řešením je tzv. balancer, tedy program který přesouvá bloky z počítačů, kde je bloků příliš na ty, kde jich je málo. Zapíná se příkazem `start-balancer.sh` a skončí, až dosáhne zadaného „prahu“ (threshold), zadávaného v procentech diskové kapacity (impicitně 10%). Případně lze vypnout příkazem `stop-balancer.sh`.

3.3.3 Přístupová práva

Přístupová práva jsou velice podobná těm ze standardu POSIX s výjimkou práva spuštění (`x`) pro soubor, které se ignoruje, neboť není možné spouštět soubory z HDFS.

Implicitně je identita uživatele určena uživatelem a skupinami procesu (konkrétně z příkazů `whoami` a `groups`). Protože klienti jsou vzdálení, je možné se stát jakýmkoli uživatelem. Přístupová práva jsou tedy implicitně pouze ochrana před nechtěným přepsáním dat, ale nikoli před zneužitím. Jinak řečeno zajišťují autorizaci, ale autentizace implicitně zajištěna není. Lze ji však nastavit pomocí systému Kerberos.

Kontrola přístupových práv může být úplně vypnuta (volbou `dfs.permissions = false`).

3.3.4 Aplikační rozhraní

Hadoop je napsaný v jazyce Java a všechny interakce s jeho souborovým systémem (ne nutně HDFS) jsou zprostředkovány skrze Java API[3]. Naproti tomu HDFS samotný poskytuje mnoho rozhraní:

Thrift pro C++, PHP, Perl, Python, Ruby, ...

C poskytuje knihovnu *libhdfs* jayzka C.

HTTP rozhraní „pouze pro čtení“ využívané webovým serverem démonu namenode.

FUSE dovoluje připojit distribuovaný souborový systém jako souborový systém UNIXu.

3.4 MapReduce v systému Hadoop

3.4.1 Architektura

Implementace se skládá ze čtyř nezávislých entit:

Klient je jakýkoli počítač připojený ke clusteru, kde metoda `JobClient.runJob(conf)` spustí úlohu (job). Ta (metodou `submitJob()`) požádá jobtracker o nové ID, zkontroluje specifikaci výstupu (např. jestli výstupní adresář již neexistuje), spočítá rozdělení vstupu (tzv. input splits, tedy logické rozdělení souborů HDFS – jeden split je seznam umístění daných dat v HDFS a jejich délka), zkopíruje zdroje požadované úlohou (JAR soubor úlohy, konfigurační soubor úlohy a spočítané rozdělení) do souborového systému jobtrackeru (zpravidla HDFS) a řekne jobtrackeru, že úloha je připravena ke

3. V budoucnu by měl Hadoop zavést Avro, které umožní psát klienty i v jiných jazycích než je Java.

spuštění. Dále úloha zobrazuje stav výpočtu v procentech zvlášť pro funkce `map` a `reduce`.

Jobtracker je označení démonu a počítače na kterém běží. Démon koordinuje běh úlohy spouštěním úkolů (task) na které je úloha rozdělena. Řídí se komunikací s tasktrackery a podle zvoleného plánovacího algoritmu (implicitní je FIFO s nastavitelnou prioritou prostřednictvím volby mapred.job.priority nebo metody `JobClient.setJobPriority()` na `VERY_HIGH, HIGH, NORMAL, LOW, VERY_LOW`; další jsou Fair Scheduler – spravedlivé sdílení celého clusteru uživateli a Capacity Scheduler – simulující pro každého uživatele nebo organizaci oddělený cluster). Zpravidla běží na stejném počítači jako namenode (tj. řídícím). Jeho webové rozhraní je na portu 50030.

Tasktracker je označení démonu a počítače na kterém běží. Démon provádí úkoly (task – neboli podúlohy na které je rozdělena úloha). Pravidelně posílá jobtrackeru tzv. heartbeat, tedy potvrzení, že tasktracker funguje, jehož součástí je sdělení, jestli je připraven spustit nový úkol. Počet funkcí *map* a *reduce*, které může spustit je ohraničen těmito volbami:

```
mapred.tasktracker.map.tasks.maximum
mapred.tasktracker.reduce.tasks.maximum
```

Obě mají implicitní hodnotou 2. Implicitně dostane každý úkol 200 MB paměti (volba `mapred.child.java.opts` implicitně `-Xmx200m`). Zpravidla běží na stejných počítačích jako datanode.

Distribuovaný souborový systém (zpravidla HDFS – viz. 3.3), kde ostatní entity sdílí zdroje požadované úlohou (tedy JAR soubor úlohy, konfigurační soubor úlohy a spočítané rozdělení). Např. JAR soubor je pro snadnou dostupnost replikován na 10 kopií (nastavitelné v `mapred.submit.replication`).

3.4.2 Programování

Základní API je přirozeně v jazyce Java. Nevýhodou tohoto API je jeho nízkoúrovňovost a zbytečná rozsáhlost psaného kódu. Například pseudokód výpočtu četností slov pro schéma MapReduce má 8 řádků:

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

(převzato z [4] kapitola 2.1)

Kód v API systému Hadoop je ale daleko delší (viz příloha A) – 46 neprázdných řádků. I když odečteme hlavičku (`package`, `import`) a počítáme jen řádky končící „`;`“ nebo „`{`“ dostaneme 29 řádků. Tento nedostatek systému Hadoop řeší různé nástavby (Streaming, Dumbo, Pig, Pydoop, Cascading, Hive, Scoobi, Rhipe, ...). Dobrým vodítkem při výběru může být webová stránka „Looking for a map reduce language“[4] na blogu Antonia Piccolboniho.

3.5 Konfigurace

Konfigurační soubory jsou uloženy v adresáři specifikovaném proměnnou prostředí `$HADOOP_CONF_DIR`.

Soubor	Formát	Popis
hadoop-env.sh	skript v jazyce Bash	Proměnné prostředí používané ve skriptech spouštějících Hadoop.
core-site.xml	Hadoop configuration XML	Nastavení jádra Hadoopu, volby jako vstupně/výstupní nastavení společné pro HDFS a MapReduce.
hdfs-site.xml	Hadoop configuration XML	Nastavení démonů HDFS (namenode, datanode, secondary namenode – viz 3.3.1).
mapred-site.xml	Hadoop configuration XML	Nastavení démonů MapReduce (jobtracker, tasktracker – viz 3.4.1).
masters	čistý text	Seznam strojů (jeden na řádek), na kterých běží secondary namenode.
slaves	čistý text	Seznam strojů (jeden na řádek), na kterých běží datanode a tasktracker.
hadoop-metrics.properties	Java Properties	Nastavení způsobu výpisu tzv. metrik (údajů, které si ukládají démony – datanode např. ukládá počet zapsaných bajtů, ...).
log4j.properties	Java Properties	Nastavení systémových souborů log, tzv. souboru HDFS audit log a soubory log podúloh tasktrckeru (viz 3.6)

(Přeloženo z [6] Table 9-1 str. 266.)

Seznamy voleb s implicitními hodnotami jsou pro verzi Hadoop 0.20 a jednotlivé XML soubory na adresách:

core-site.xml `http://hadoop.apache.org/docs/r0.20.2/core-default.html`

hdfs-site.xml `http://hadoop.apache.org/docs/r0.20.2/hdfs-default.html`

mapred-site.xml `http://hadoop.apache.org/docs/r0.20.2/mapred-default.html`

Nebo ve stejnojmenných souborech v adresáři `$HADOOP_HOME/docs`

Volby se zapisují do XML souborů (pod kořenovou entitu `<configuration>`) např. takto:

```
<property>
        <name>dfs.name.dir</name>
        <value>/var/tmp/hadoop-${user.name}/name</value>
</property>
```

3.6 Soubory log

Pokud v systému Hadoop selže úloha, nebývá lehké najít příčinu. Chyba může nastat nevhodnou konfigurací nějakého démonu, chybným zadáním úlohy, nebo třeba vyčerpáním

4. `http://blog.piccolboni.info/2011/04/looking-for-map-reduce-language.html`

místa na disku či v paměti. Může se tedy stát, že řídící se dozví pouze, který démon kterého uzlu přestal běžet, nebo na kterých uzlech selhalo provádění úkolu (task). Samotnou příčinu je ale nutné hledat podřízených uzlech. Je tedy dobré znát základní logy, kde je možné problém hledat.

Logy systémových démonů: démony systému Hadoop (zmiňované namenode, datanode, jobtracker a tasktracker, ale stejný princip platí např. i pro HBase) používají log4j,jehož výstup nalezneme v souboru se jménem démonu v názvu a s koncovkou `.log` v adresáři `$HADOOP_LOG_DIR`. V témž adresáři je i stejnojmenný soubor s koncovkou `.out`, kam se ukládá chybový a standardní výstup démonu. Ten je ale zpravidla je prázdný.

Logy úkolů: úkol (task, neboli část úlohy řešená na podřízeném) zapisuje do 3 souborů: log4j do souboru `syslog` a standardní výstupy do souborů `stdout` a `stderr`. Tyto soubory jsou na podřízených uzlech v adresáři `$HADOOP_LOG_DIR/userlogs`. Například při použití Hadoop Streaming (více v 6.6), je chybový výstup spouštěného externího programu tady v souboru `stderr`.

Kapitola 4

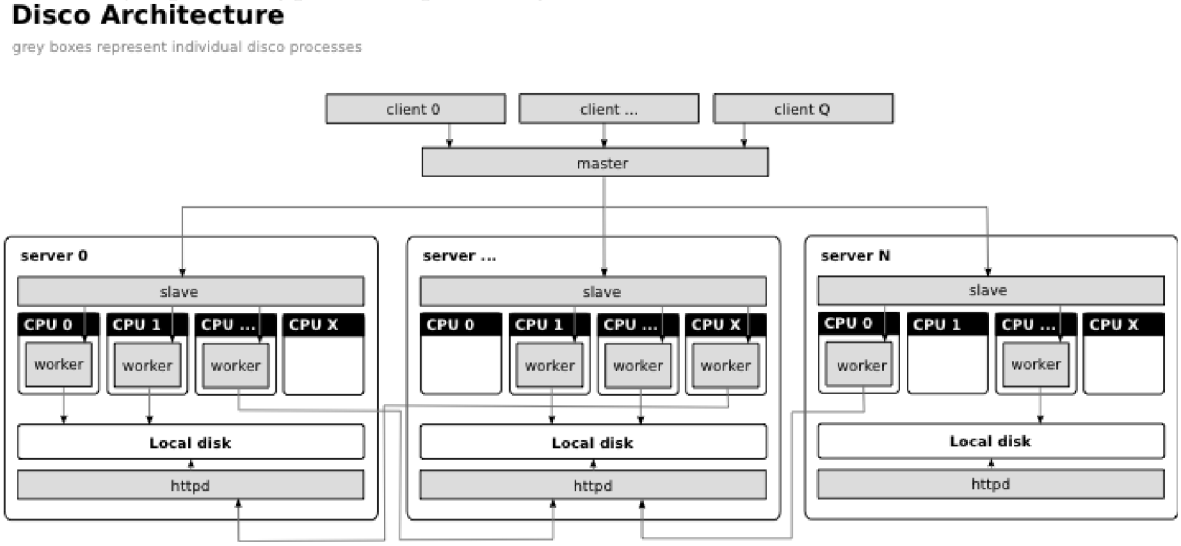
Disco

Disco je volně dostupný framework pro distribuované výpočty založený na MapReduce. Vznikl v Nokia Research Center a jeho moto je „massive data – minimal code“. Jádro frameworku je napsané v jazyce Erlang[1]. Pro psaní samotných aplikací MapReduce se však zpravidla používá jazyk Python. Existuje ovšem například alternativa ODisco, která poskytuje rozhraní v jazyce OCaml.

4.1 Architektura systému Disco

Tato sekce čerpá z `http://disco.readthedocs.org/en/0.4.5/overview.html`.

Disco je aplikace typu řídící/podřízený (master/slave):



Master je démon, který přijímá úlohy, přidává je do fronty a spouští je na clusteru, když jsou uzly dostupné.

Client je program, který posílá úlohy na master.

Slave je démon spuštěný masterem na každém uzlu v clusteru. Vytváří a monitoruje všechny procesy, které běží na daném clusteru.

Worker je program, vykonává úkoly, ze kterých se skládá úloha. Umístění výstupu je odesláno na master. Více v sekci 4.3.2.

1. funkcionální programovací jazyk vyvinutý firmou Ericsson pro podporu distribuovaných, chybám odolných, nepřetržitě běžících aplikací

Disco se snaží, aby úkoly běžely – pokud možno – na lokálních datech. Každý uzel má ovšem server HTTP, který umožňuje přistupovat k datům vzdáleně.

Uživatelé mohou jednoduše nastavit počet úkolů, které běží paralelně na jednom uzlu (např. na počet jader, nebo na počet disků). Dokonce lze jádra na jednom clusteru rozdělit mezi více masterů, což minimalizuje možnost naprostého selhání v případě selhání masteru.

4.2 Disco Distributed File System

Tato sekce popisuje základní rysy distribuovaného souborového systému vyvinutého v rámci projektu Disco. Více informací je na webové stránce [1].

DDFS je distribuovaný souborový systém navržený pro potřeby systému Disco a schématu MapReduce obecně. Není to systém kompatibilní se standardem POSIX, ale – podobně jako HDFS (viz 3.3) nebo GFS (Google File System) – speciální vrstva nad souborovým systémem operačního systému. Její funkce jsou distribuce, replikace, persistence a adresování dat, a také řízení přístupu k nim. Její základní příkazy lze zadávat pomocí příkazu `ddfs` např. takto:

```
ddfs ls [argument]
```

Základem DDFS ovšem nejsou soubory a adresáře, ale tzv. bloby a tagy:

Blob (Binary large object) je část dat, kterou Disco distribuuje po síti s pod určitou adresou URL (např. `disco://nymfe10/ddfs/vol0/blob/ba/ae_10MB-0$559-16f56-93d4e` kde `disco://nymfe10/` znamená zpravidla `http://nymfe10:8989/`). Každá replika má svou URL.

Tag obsahuje metadata o blobech a zejména seznam adres URL. Každý tag má také svou URL např. `disco://alba/ddfs/tag/inputs:ae_10MB`, tedy tag může obsahovat i odkazy na další tagy. Tag může obsahovat ještě atributy (UTF-8 řetězce), jejichž speciálním případem může být autorizační token HTTP.

Název tagu, blobu i atributu musí odpovídat regulárnímu výrazu `[A-Za-z0-9_\-@:]+`. Zejména se tedy nedají použít znaky „.“ a „/“. Tagy nejsou rovnány do hierarchické struktury, ale pomyslnou hierarchii mohou tvořit jejich názvy, ve kterých se na místo „/“ používá „:“ (např. `user:xnovak938:data`). Volání výpisu tagů totiž berou za parametr předponu (`$ ddfs ls [prefix ...]`).

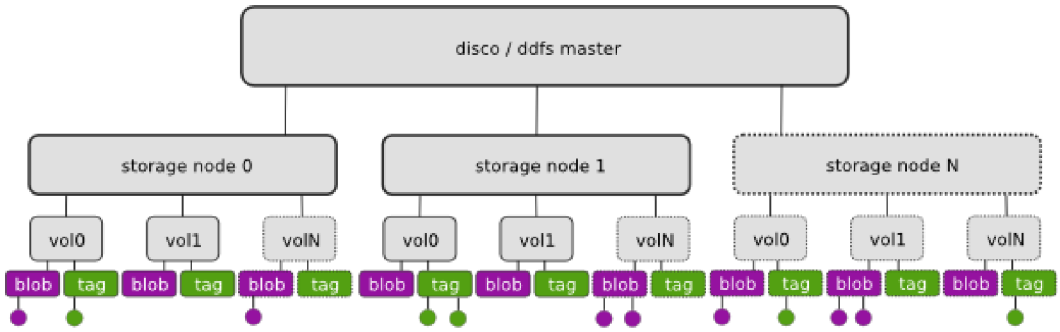
DDFS je určený pro ukládání nestrukturovaných dat, ale nehodí se pro data s velmi malými položkami (méně než 4K). Je horizontálně škálovatelný (za běhu mohou být přidávány nové uzly) a je odolný vůči chybám díky replikaci.

4.2.1 Aplikační rozhraní

Základní aplikační rozhraní tvoří metody protokolu HTTP. Toto rozhraní je využito v modulu `disco.ddfs`, který poskytuje rozhraní pro Python. Rozhraní Pythonu je zase použito příkazem `ddfs`, jež poskytuje rozhraní pro příkazovou řádku.

4.2.2 Architektura a implementace

DDFS nemá samostatný démon, ale je součástí systému Disco. Každý uzel obsahuje určitý počet svazků přiřazených do DDFS připojením k adresářům `DISCO_ROOT/vol0...DISCO_ROOT/volN`[2]. Na každém svazku DDFS ukládá data do adresářů `blob` a `tag`. DDFS monitoruje volné místo na svazcích a nové bloby ukládá na nejméně plné uzly.



4.2.3 Uložení dat

DDFS (narozdíl od HDFS) neukládá data do bloků pevně stanovené velikosti. Místo toho se snaží vytvořit hranice bloků na hranicích záznamů, což potom zjednodušuje práci s daty. Daní za tuto výhodu je ovšem značné zpomalení nahrávání (viz 7.1).

DDFS ukládá data buď v původním formátu, nebo v interním komprimovaném formátu. Pokud chce uživatel uložit výsledek MapReduce úlohy do DDFS (parametr `save=True` v metodě `Job.run()`), použije se interní formát. Nahrávat data na cluster lze dvěma způsoby:

1. `ddfs push tag [file ...]` a `ddfs put [url ...]` nahrají data v původním formátu a nerozdělelná. Velké soubory je tedy dobré rozdělit (např. z příkazové řádky příkazem `split` nebo `csplit`), aby je bylo možné zpracovat paralelně.

2. `ddfs chunk tag [url ...]` nahrává data po částech (implicitně 64MiB velkých) ve vnitřním komprimovaném formátu, který komprimuje data na přibližně na polovinu. Důležitou volbou je `-R reader`, kde uživatel může implementovat svou metodu `disco.worker.classic.func.input_stream`, která zajistí rozdělení dat tak, jak potřebuje.

4.3 Distribuce úloh

Úlohy systému Disco jsou distribuovány pomocí objektu Job Pack. Job Pack obsahuje vše, co je potřeba pro běh úlohy, zejména spustitelný kód úkolu pro podřízený stroj, tzv. **worker**. Před spuštěním je tento objekt zkopírován pomocí HTTP z řídícího uzlu na podřízené.

2. kde DISCO_ROOT je kořenový adresář DDFS

4.3.1 Job Pack

Job Pack obsahuje:

Job Dict – slovník formátu JSON:

```
{ "input"      : [ [replica_1_URL, replica_2_URL, replica_3_URL ], ...],
  "worker"     : "relative/path/to/worker/binary",
  "map?"       : true | false,
  "reduce?"    : true | false,
  "nr_reduces" : <Non-negative integer>,
  "prefix"     : "unique_string",
  "scheduler"  : { "max_cores" : <Non-negative integer>,
                   "force_local": true | false,
                   "force_remote": true | false
                 },
  "owner"      : "user_name"
}
```

Job Environment Variables – slovník formátu JSON s proměnnými prostředí.

Job Home – zazipovaný adresář se soubory, které potřebuje worker.

Additional Job Data – libovolná dodatečná data. Ve standardní implementaci obsahuje serializovaný worker.

Podrobnosti na stránce `http://disco.readthedocs.org/en/0.4.5/howto/jobpack.html`.

4.3.2 Worker

Worker je buď spustitelný binární kód, nebo skript provádějící jiné spustitelné kódy, zadaný řídícím uzlem. Worker komunikuje se systémem Disco pomocí protokolu Disco Worker Protocol. Worker iniciuje komunikaci zprávou a dostává odpovědi. Zprávy jsou posílány přes standardní chybový výstup a odpovědi přijímány na standardním vstupu. Standardní výstup bývá přesměrován na chybový, takže worker by na ně neměl nic zapisovat, kromě zpráv protokolu.

Formát zprávy – jak od řídícího uzlu, tak od podřízeného – je:

```
<name> 'SP' <payload-len> 'SP' <payload> '\n'
```

kde `<name>` je jeden z řetězců `DONE`, `ERROR`, `FAIL`, `FATAL`, `INPUT`, `INPUT_ERR`, `MSG`, `OK`, `OUTPUT`, `PING`, `RETRY`, `TASK`, `WAIT`, `WORKER`; `'SP'` je jedna mezera, `<payload>` objekt v notaci JSON a `<payload-len>` jeho délka v bajtech.

Podrobnosti na stránce `http://disco.readthedocs.org/en/0.4.5/howto/worker.html#worker-protocol`.

4.4 MapReduce v systému Disco

Ve standardní implementaci zajišťují distribuci úloh (popsanou v sekci 4.3) třídy disco.job.JobPack a disco.worker.Worker. Implementaci MapReduce potom podtřída zmíněné třídy Worker: disco.worker.classic.worker.Worker.

4.4.1 Programování

Typické spuštění úlohy vypadá takto:

```
from disco.job import Job
results = Job(name).run(**jobargs).wait()
```

Třída Job instanciuje Worker, metoda run vytvoří JobPack a metoda wait čeká, dokud úloha neskončí.

Základní příklad na četnost slov vypadá takto:

```
from disco.core import Job

def map(line, params):
    for word in line.split():
        yield word, 1

def reduce(iter, params):
    from disco.util import kvgroup
    for word, counts in kvgroup(sorted(iter)):
        yield word, sum(counts)

if __name__ == '__main__':
    job = Job().run(input=["http://address.of/input.txt"],
                    save=True,
                    map=map,
                    reduce=reduce).wait()
```

Definice funkcí map a reduce velmi připomínají pseudokód úlohy zmíněný v kapitole 3.4.2 a zbytek kódu není dlouhý, takže lze říct, že Disco má přehledné API, mnohem přehlednější než základní API systému Hadoop.

Kapitola 5

Instalace systémů MapReduce v CZPJ

CZPJ používá pro své výpočty výkonné servery (alba, aura, asteria04) a dalších několik méně výkonných počítačů. FI MUNI má ovšem mnoho počítačů, na kterých sice probíhá výuka, ale v nočních hodinách a o víkendech zůstává jejich potenciál většinou nevyužit. Tato práce se snaží zapojit do výpočtů CZPJ ještě počítače nymfe, a zároveň co nejlépe využít jejich výpočetní potenciál dohromady s potenciálem zmíněných serverů.

Systémy MapReduce jsou instalovány do systému modulů, fakultní sítě, který je sdílený se sítí CZPJ. Jejich konfigurace je uvedena v domovském adresáři uživatele hadoopnlp, pod kterým také běží jednotlivé démony.

5.1 Počítače nymfe

Fakultní síť obsahuje přes 60 počítačů nymfe s nainstalovaným Linuxem. Po dobu práce na této diplomce platilo, že většina jich měla dvě jádra procesoru a bylo potřeba počítat i se čtyřjádrovými. V době provádění zde uvedených výpočtů ale už byly čtyřjádrové všechny. Počítače jsou volně přístupné studentům, někdy na nich probíhá výuka a na noc se vypínají, pokud na nich není nikdo aktivně přihlášený. Pro experimenty s paralelním počítáním se tedy hodí zabrat vždy jen ty stroje, na kterých není nikdo přihlášený, popř. je tam přihlášený jeden uživatel, který brání jejich večernímu vypnutí. Někdy je pro rozhodnutí, zda zařadit uzel do clusteru dobrý obecně jakýkoli příkaz. U vybraných počítačů se může hodit vědě, kolik mají jader. Výpis počítačů vhodných podle těchto kritérií zajišťuje příkaz `available_nymfes.sh`, který má k dispozici uživatel hadoopnlp v adresáři `/home/hadoopnlp/bin`:

```
hadoopnlp@alba:~$ available_nymfes.sh -h
Print available nymfe nodes (possibly only those with runable COMMAND which is executed)
Usage: /home/hadoopnlp/bin/available_nymfes.sh [-a|-b][-c][-f][COMMAND]
   -a       only hall
   -b       only B130
   -c       print number of cores
   -f       only free nodes (those with empty `who`)
   -o USER  free nodes or those occupied only by USER
   COMMAND: any bash command e.g. [ -d ~/ ] for the nodes with available HOME
```

5.2 Hadoop

Použitá verze je `hadoop-0.20.2-cdh3u1`. Uživatel hadoopnlp má k dispozici standardní příkazy systému hadoop, z nichž za zmínku stojí:

hadoop-daemon.sh který rozběhne nebo zastaví zadaný démon (namenode, secondarynamenode, jobtracker, datanode, tasktracker) na uzlu, kde je zadán.

start-all.sh který rozběhne namenode a jobtracker na uzlu kde je zadán, na uzlech uvedených v `$HADOOP_CONF_DIR/masters` rozběhne secondarynamenode a na uzlech uvedených v `$HADOOP_CONF_DIR/slaves` rozběhne datanode a tasktracker.

stop-all.sh všechny démony zastaví.

Pomocí těchto příkazů lze rozběhnou Hadoop podle implicitní konfigurace (adresář `/home/hadoopnlp/conf`). Protože ovšem cluster není sourodý (některé počítače mohou mít 2 jádra, některé mají 4, výpočetní servery mají úplně jiný počet jader) a všechny uzly jsou uvedeny v jednom konfiguračním souboru `slaves`, je nejjednodušší po startu clusteru ještě restartovat servery a čtyřjádrové stroje nymfe (většina je je dvoujádrových) s jinou konfigurací. K tomu v rámci práce vznikly pro uživatele hadoopnlp příkazy `restart_sc.sh`, `restart_4core.sh`.

Adresář `/home/hadoopnlp/bin` také obsahuje skript `racks.sh`, který pomáhá Hadoopu mít cluster rozdělený do „stojanů“ tak, aby uzly alba, asteria04 a aura mohly být zvlášť a udržovat v chodu výpočet i potom, co je potřeba vypnout Hadoop na počítačích nymfe.

V neposlední řadě obsahuje `/home/hadoopnlp/bin` skript `reset_hadoop.sh`, který smaže všechny data a restartuje cluster (což někdy může být nejrychlejší řešení, pokud se data v clusteru dostanou do nekonzistentního stavu).

5.3 Disco

Konfigurační soubor je /home/hadoopnlp/.disco, ale konfiguraci clusteru obsahuje soubor `/var/tmp/disco_root/disco_8989.config` na počítači zvoleném za master (tedy tom, kde bylo spuštěno `disco_start.sh`). Disco se dá velmi pohodlně konfigurovat přes webové rozhraní typicky na portu 8989. Pro nejčastější požadavek zahrnutí dostupných počítačů nymfe s počtem jader odpovídajícím jejich procesoru je však po uživatele hadoopnlp lepší příkaz:

```
available_nymfes.sh -c | disco_config.sh > /var/tmp/disco\_root/disco\_8989.config
```

Pro fungování Disca je nutný soubor `/home/hadoopnlp/.erlang.cookie` na všech uzlech a také adresář `/var/tmp/disco_root/ddfs`. Chyby, jako je absence `/var/tmp/disco_root/ddfs` signalizuje webové rozhraní červenou barvou na daném uzlu.

Kapitola 6

Zpracování velkých dat v CZPJ

Korpusy v CZPJ se ukládají v textových souborech velkých až stovky GB. Zpracovávají se „proudově“ (úpravou sekvenčně čtených dat), neboť jejich editace v běžných editorech postrádá kvůli velikosti smysl. K tomu účelu se osvědčil jazyk příkazové řádky systému UNIX. Zejména jeho schopnost řetězit programy typu roura[1]. Jednodušší úlohy lze tedy řešit spojením příkazů jako jsou `tr`, `cut`, `sed`, `awk`, `...`; ty složitější se osvědčuje implementovat v nějakém skriptovacím jazyce právě jako rouru. Zpracovat velká data na jednom počítači je ale pomalé. Korpusy bývají ovšem rozděleny (zpravidla SGML značkami `<doc>`, `</doc>`) na dokumenty, které lze zpracovávat zvlášť. To umožňuje jednoduchou paralelizaci.

6.1 Možnosti paralelizace

1. Rozdělit data, zkopírovat je na jednotlivé stroje a spustit na nich zpracování.
2. Najít bajtové pozice jednotlivých dílů vstupu, poslat tyto pozice na stroje, které sdílí soubor přes NFS a spustit tam zpracování daných dílů.
3. Použít BashReduce, který rozdělí data podle hodnot na jednotlivých řádcích mezi jednotlivé stroje, tam hodnoty seskupit a provést funkce *reduce*.
4. Skladovat data distribuovaně a zpracovávat je pomocí MapReduce.

První možnost je velmi přímočará. Oproti frameworkům MapReduce má sice menší provozní režii, ale protože uzly nedrží v kontaktu démony, mohou nastat zpoždění při navazování spojení. Zřetelnou nevýhodou je ovšem kopírování dat. Dále tento přístup neposkytuje fázi roztřídění (shuffle), takže pro úlohy vyžadující reduce může implementovat pouze instance spouštějící *combiner* a vše nakonec slít jednou instancí spouštějící *reduce*.

U druhé možnosti odpadá kopírování vstupních dat výměnou za zátěž serveru a za nároky na šířku pásma. Pokud vybavení obstojí, může tato metoda úspěšně paralelizovat zejména jednodušší typy úloh.

Třetí možnost je doménou funkcí vyžadujících reduce a tady může být rychlejší než předchozí přístupy díky roztřídění vstupu (shuffle).

Čtvrtá možnost (MapReduce) zpracovává data co nejvíce lokálně a kopírování odpadá díky distribuovanému souborovému systému. Díky replikaci je toto řešení daleko odolnější vůči chybám. Implementace ale není úplně přímočará – je třeba použít framework, který implementuje MapReduce. To znamená dodatečnou režii a může to (v závislosti na použitém rozhraní) ztížit psaní programů, nebo nutit uživatele učit se nový programovací jazyk.

1. Neboli pipeline – tedy příkazy které čtou vstup z tzv. standardního vstupu a zapisují na tzv. standardní výstup, jež může být standardním vstupem dalšího programu.

6.2 Paralelizace pomocí kopírování a SSH

Tato koncepce může mít hodně podob. V rámci této práce je implementována takto: kopírování je provedeno pomocí `bdfs.sh put` do určitého adresáře na stroje uvedené v `BDFS_NODES` (např. `BDFS_NODES=(nymfe01:4 nymfe64:2)`), vždy po tolika dílech, po kolika je za „:“ uvedeno jader. Skript `parapipe.sh` potom na daných uzlech souběžně spustí program typu roura na všech souborech zadaného adresáře, a to na všech uzlech. Výstup uloží to stejnojmenných souborů zadaného výstupního adresáře. Pokud je úloha vyžaduje funkce *reduce*, provedou se na jednom počítači. Jednoduše lze celý výstup setřídt a provést *reduce*, takto:

```
$ bdfs.sh cat output_dir/\\\* | sort | reduce_script.sh
```

Výhodnější ale je zakončit rouru distribuovanou v `parapipe.sh` tříděním a eventuálně instancí provádějící *combiner*. Potom není nutné výstup třídit, ale stačí slít jednotlivé výstupní soubory a provést *reduce*, takto:

```
$ bdfs.sh get output_dir tmp_dir && sort -m tmp_dir/* |
                    reduce_script.sh
```

6.3 Paralelizace pomocí NFS a SSH

V této koncepci předpokládáme, že vstupní soubor sdílíme na všech uzlech clusteru pomocí implementace NFS (protokolu pro sdílení vzdálených svazků). V této práci přístup implementuje skript `nfspipe.sh`, který spouští na vzdálených uzlech zadaný program typu roura vždy na jiné části stejného sdíleného souboru. Výstup je ukládán do BDFS. Úlohy, které potřebují i reduce se řeší stejně jako v případě skriptu `parapipe.sh` (viz 6.2).

6.4 BashReduce

V roce 2009 představil zaměstnanec firmy last.fm Erik Frey jednoduchý program jménem BashReduce. Je napsaný převážně v jazyce Bash a může používat pouze Bash, ale pro lepší efektivitu využívá dva krátké programy v jazyce C pro rozdělení dat (fáze shuffle) a jejich slévání.

Skript čte řádky standardního vstupu, v nichž jeden sloupec (defaultně ten první) je klíč. Skript rozdělí řádky vstupu mezi jednotlivé stroje – stejné klíče vždy na ten samý stroj. Na cílovém stroji se řádky setřídí podle klíče a provedou se Funkce *reduce* implementované jako roura poskytnutá v parametru `-r`. Typicky je to příkaz `uniq -c`.

Název BashReduce je tedy přiléhavý i proto, že skript neposkytuje možnost paralelního provedení funkcí *map*. Funkce *map* může provést pouze jedna instance implementovaná jako roura na vstupu.

Z uvedených přístupů se BashReduce nejvíce blíží „SSH a NFS“, protože jeho úzkým hrdlem může být čtení vstupu z jednoho disku. Narozdíl od „SSH a NFS“ ale může být úzkým hrdlem i nedistribuovaná instance provádějící *map*. Výhoda BashReduce ovšem je, že nepotřebuje zápisy mezivýsledků na disk, protože je zapisuje do pojmenovaných rour, propojených v síti procesy příkazu `netcat`. Navíc je jeho závěrečné slití výsledků rychlejší, díky počátečnímu roztřídění vstupu. Nevýhodou BashReduce ještě je, že při inicializaci spojení používá tolik SSH spojení, kolik je v clusteru jader, narozdíl od skriptů `parapipe.sh`

a `nfspipe.sh`, které se přihlásí vždy jednou na každý uzel. BashReduce spouští přihlášení SHH jedno po druhém, takže jeho inicializace je pomalá. To se dá v jeho kódu vcelku triviálně změnit na souběžné přihlašování mnoha podprocesy, ale ani to není ideální řešení, protože u uzlů s velkým počtem jader (jako je např. aura) hrozí v takovém případě odmítnutí spojení.

6.5 Bash Distributed File System (BDFS)

BDFS vznikl v rámci této práce, aby zjednodušil distribuci a správu souborů rozdělených mezi více uzlů clusteru jenom pomocí příkazové řádky. Spíše než plnohodnotným distribuovaným souborovým systémem se snaží být srozumitelným skriptem, pro práci se soubory uloženými na více počítačích s různými počty jader.

Na rozdíl od distribuovaných souborových systémů nemá BDFS replikaci, a tím není odolné vůči selhání nějakého uzlu. Zároveň přidání dalšího uzlu znamená, že pokud ho chci používat pro již uložená data, musím je na cluster uložit znovu. Výhodou BDFS je jeho funkcionalita, která využívá Bash, a zároveň autorizace a autentizace, která je např. v HDFS vcelku složitá, zatímco BDFS ji využívá na úrovni operačního systému.

BDFS má 5 speciálních příkazů a jinak umožňuje používat víceméně všechny příkazy jazyka Bash. Jeho součástí je konfigurační soubor, který specifikuje zejména kořenový adresář (`BDSF_ROOT`), ve kterém se budou nacházet data na cílových uzlech, a které uzly a kolik jejich jader se bude používat (`BDFS_NODES`).

Speciální příkazy jsou:

`bdfs.sh get bdfs_dir local_dir` stáhne obsah adresářů `BDFS_ROOT/bdfs_dir` ze všech uzlů do adresáře `local_dir`.

`bdfs.sh put [-d DELIM_RE] local_file bdfs_dir` rozdělí soubor na tolik dílů, kolik je v clusteru jader. Zkopíruje díly do adresářů `BDFS_ROOT/bdfs_dir` jako soubory pojmenované `000`, `001`...`NNN` (nebo jinak podle `BDFS_FILE_NAMES`).

`bdfs.sh putdir [-d DELIM_RE]local_dir bdfs_dir` spustí `bdfs.sh put` pro každý soubor v adresáři `local_dir` (nikoli však ve stromu adresářů).

`bdfs.sh firsthead [HEAD_OPTIONS]bdfs_dir` spustí `head` pouze na prvním souboru v `bdfs_dir` prvního uzlu.

`bdfs.sh lasttail [TAIL_OPTIONS]bdfs_dir` spustí `tail` pouze na posledním souboru v `bdfs_dir` posledního uzlu, na kterém se vyskytují v `bdfs_dir` nějaké soubory.

Příkazy jazyka Bash jsou spouštěny po jedné instanci na každém stroji. Pokud tisknou na standardní výstup, je tento uložen do dočasných souborů (na uzlu kde byl příkaz spuštěn) a ve správném pořadí vytištěn. Náročnější výpočty se vyplatí spouštět na každém jádře, ale k tomu slouží příkaz `parapipe.sh` (viz 6.2).

6.6 Paralelizace pomocí Hadoop

Spuštění programů typu roura řeší v sytému Hadoop rozšíření jménem Streaming. Protože příkazy spouštějící Streaming potřebují specifikovat moho voleb a ve výsledku zabere příkaz

třeba několik řádků, přichází tato práce se skriptem `hadpipe.sh`, který spouští Streaming, ale umožňuje stručnější zadání voleb.

Při použití nadstaveb, jako je Hadoop Streaming ovšem hrozí zpomalení výpočtu. Práce proto v některých případech srovnává výpočty programu `hadpipe.sh` s řešením v základním API Systému Hadoop.

6.7 Paralelizace pomocí Disco

Disco implicitně nenabízí spouštění programů typu roura, a proto tato práce přichází se skriptem `dispipe.py`, který spouštění rour umožňuje. Rozhraní Disca je ovšem v jazyce Python, který je hojně používán v rámci CZPJ, takže u některých úloh stojí za zvážení, zda použít `dispipe.py`, nebo přímo rozhraní Disca.

Kapitola 7

Porovnání jednotlivých přístupů

7.1 Nahrávání dat na cluster

Nahrávání dat nemusí být významným hlediskem pro fungování distribuovaného systému, pokud už jsou všechna data v něm. Pokud ale daný systém není úplně zaběhnutý, může to být hledisko velice významné a je to významné hledisko pro BDFS (viz 6.5), kde se s trvalým uložením dat nepočítá.

DDFS a HDFS mají k nahrávání odlišný přístup. DDFS ukládá bloky dat nepevné velikosti, tak aby nerozdělil záznamy. HDFS ukládá data do pevně velkých bloků (implicitně 64 MB) a jejich rozdělení na záznamy řeší Hadoop v rámci úlohy. To značně zpomaluje nahrávání souborů do DDFS a může to zpomalovat úlohy Hadoopu.

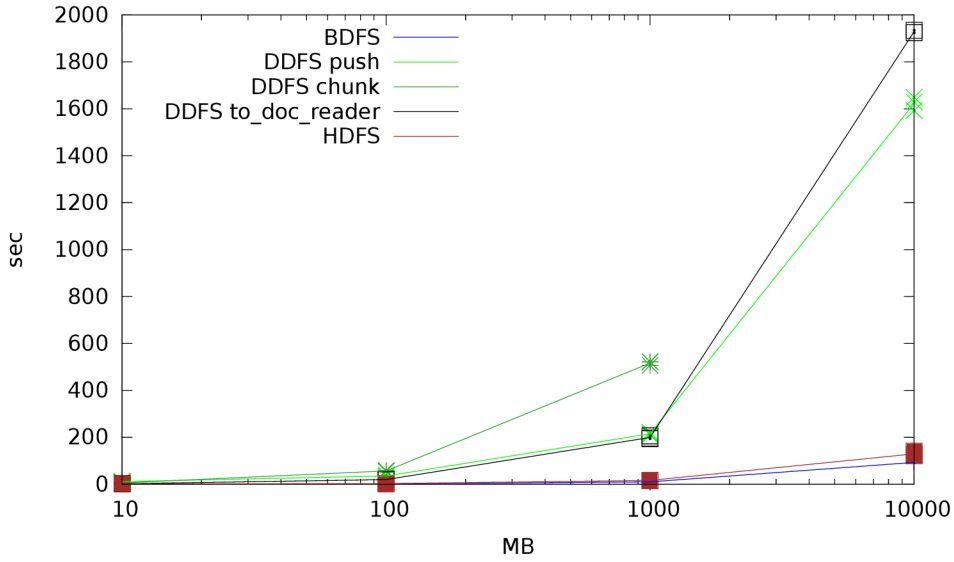
DDFS má dva způsoby nahrávání dat, DDFS push a DDFS chunk (viz 4.2.3). Další možností je uzpůsobit DDFS chunk pomocí parametru `-R reader`. DDFS chunk se může zrychlit, pokud nažítá jako jednotlivé záznamy celé dokumenty místo řádků (varianta označovaná v grafech DDFS to_doc_reader). S daty uloženými touto cestou se pak ale musí často pracovat trochu nestandardně. Navíc má Disco omezenou velikost záznamu na 1 MB (což se mimochodem dá jednoduše změnit ve zdrojových kódech).

BDFS stejně jako DDFS nerozděluje záznamy. Rozdělí soubor (pokud možno) na stejný počet dílů, jako je v clusteru jader. Aby si ušetřil čtení a zápis na disk, používá skript `split_offsets.py` (viz příloha B), který nečte celý soubor, ale pokud možno provádí `seek`, zároveň nezapisuje na disk, ale pouze vytiskne pozice.

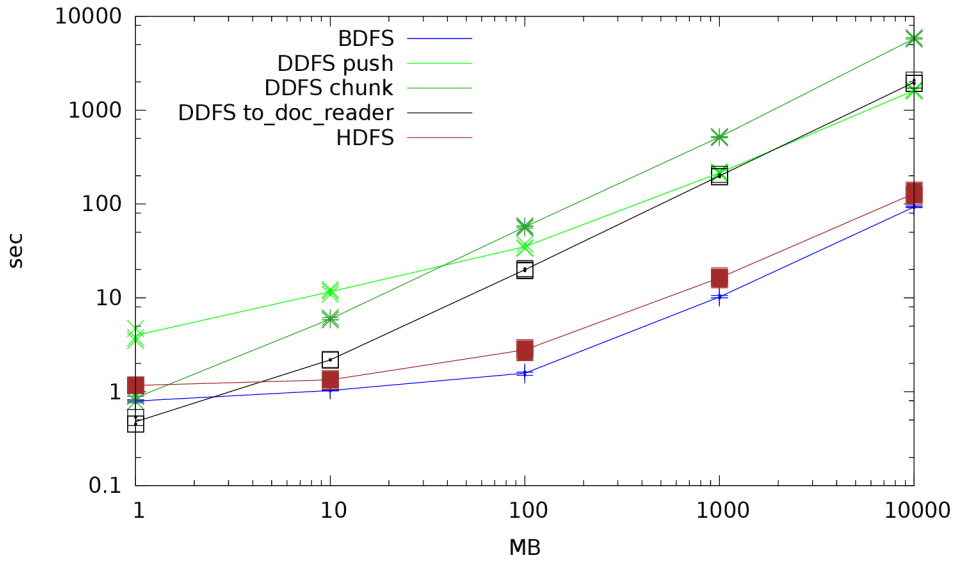
Srovnání nabízí grafy 7.1 a 7.2. Grafy ukazují hlavně na to, že DDFS je v nahrávání velmi pomalé, takže pro přehlednost chybí v 7.1 hodnoty DDFS chunk pro 10 GB (kolem 5800 s). Všechny naměřené hodnoty jsou v grafu 7.2 s logaritmickou škálou a v tabulce C.1.

Grafy také potvrzují, že DDFS chunk je oproti DDFS push mnohem pomalejší kvůli členení dat na jednotlivé řádky. Hodnoty pro DDFS to_doc_reader, (což je varianta DDFS chunk) se totiž blíží spíše hodnotám pro DDFS push.

V neposlední řadě grafy říkají, že BDFS je v nahrávání stejně rychlé nebo i trochu rychlejší než Hadoop, jakkoli toto srovnání může kulhat – třeba proto, že Hadoop kvůli replikaci musí umístit třikrát více dat.



Obrázek 7.1: Nahrávání dat ze stroje alba na stroje alba, aura, asteria04 a 62 nymfeXX.

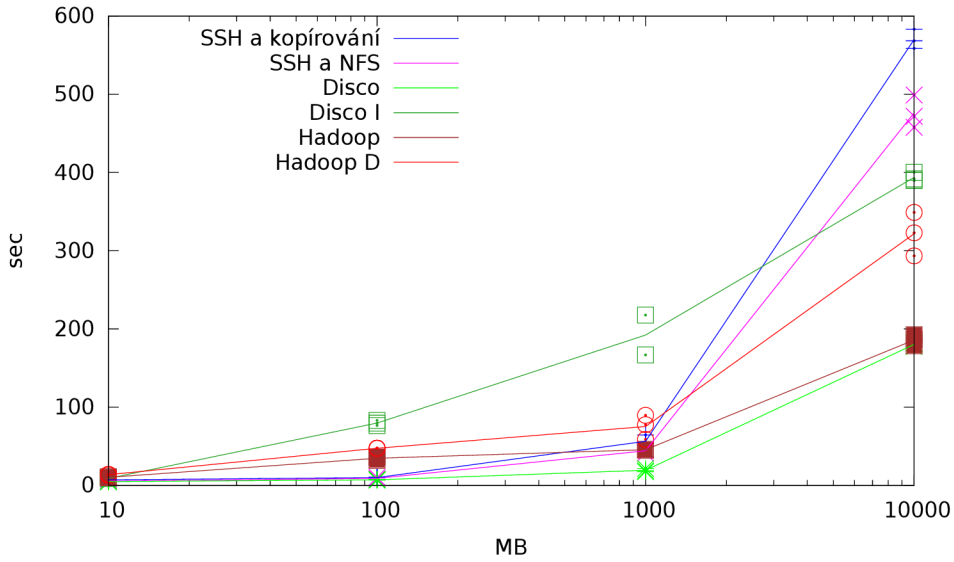


Obrázek 7.2: Nahrávání dat ze stroje alba na stroje alba, aura, asteria04 a 62 nymfeXX.

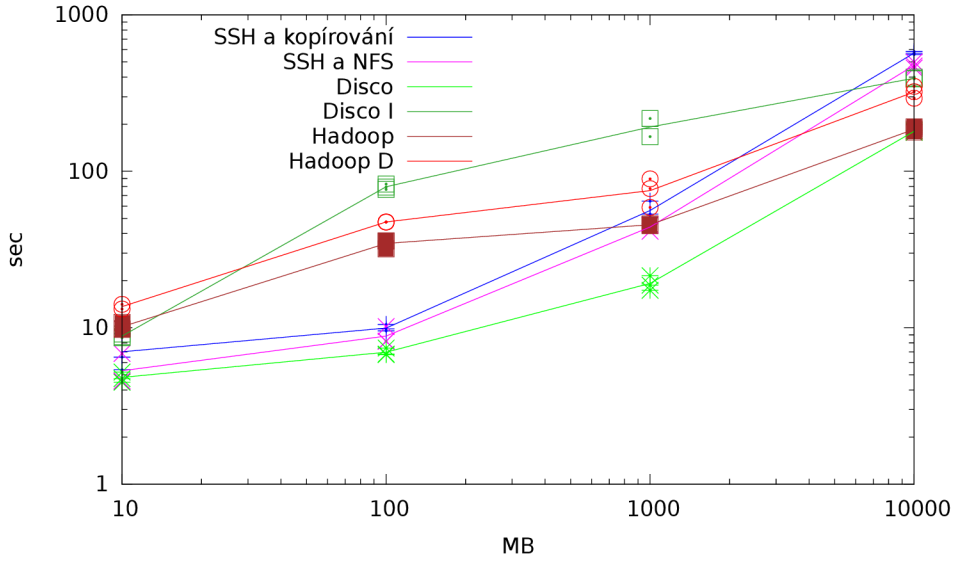
7.2 TreeTagger

Velká část netriviálních úloh NLP jsou úlohy typu map. Patří sem například značkování korpusů (ať už morfologické, či syntaktické), což je také velmi častá úloha spouštěná na počítačích CZPJ. Oblíbeným nástrojem pro morfologické značkování je statistický značkovač TreeTagger [5], úspěšně používaný nejméně pro 16 jazyků a adaptovatelný na další.

Grafy 7.3 7.4a ukazuje na TreeTaggeru pro anglické texty nasazení jednotlivých přístupů uvedených v kapitole 6. Naměřené hodnoty jsou v tabulce C.2.



Obrázek 7.3: TreeTagger na strojích alba, aura, asteria04 a 62 nymfeXX.



Obrázek 7.4: TreeTagger na strojích alba, aura, asteria04 a 62 nymfeXX.

Grafy 7.3 a 7.4 nezapočítávají kopírování do distribuovaných souborových systémů ani

stáhnutí výsledků (pouze u BDFS, které není plnohodnotný DFS, takže se tomu nelze vyhnout). Je z nich dobře vidět, že už při značkování dat 10 GB se vyplatí mít data trvale uložená v DFS nějakého frameworku.

Dále je vidět velký rozdíl mezi výsledky Disca a Disca čtoucího soubory uložené v interním formátu Disco I. To může být způsobeno dvěma jevy. Jednak je interní formát komprimovaný, což může výpočet zpomalovat. Za druhé `ddfs chunk` distribuuje data v blocích zhruba po 64 MB, zatímco běžný text je distribuovaný po malých kouscích, jež (pokud možno) odpovídá počtu jader v clusteru, takže např. soubor velký 100 MB nahraný pomocí DDFS chunk zpracovávají dva uzly (výpočty označené Disco I), zatímco stejně velký soubor nahraný pomocí DDFS push (výpočty označené Disco) zpracovávají (v závislosti na rozdělení, které provedl uživatel) třeba všechna jádra clusteru.

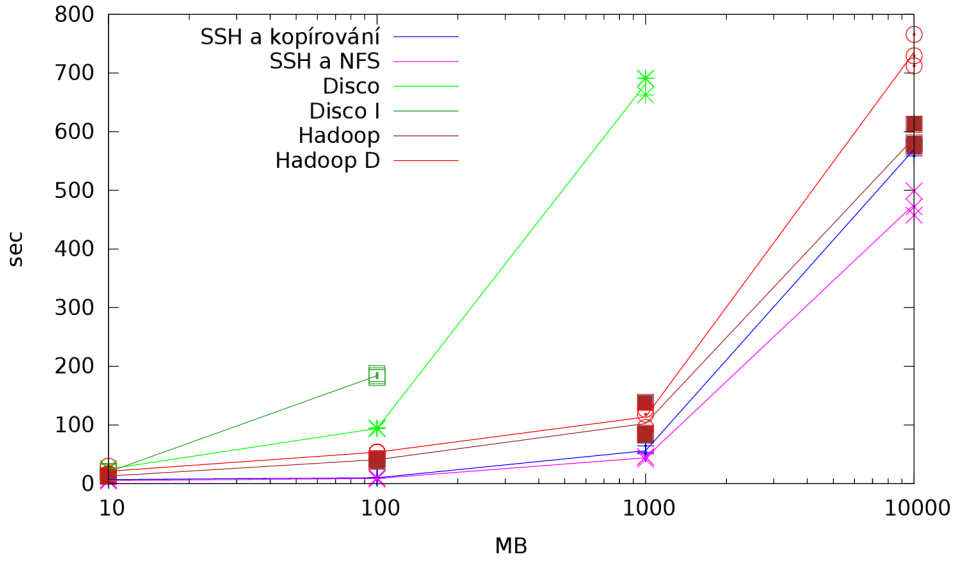
Ve dvou verzích je zde i Hadoop. Má totiž dva přístupy jak zpracovávat data tak, aby nedošlo k rozdělení záznamů. Hadoop vždy na začátku úlohy nastaví tzv. `splits`, neboli hranice ve kterých se bude pohybovat instance vykonávající *map*. Ta je ovšem může porušit, typicky proto, aby dočetla řádek. V třídě TextInputFormat lze dokonce specifikovat jakýkoli řetězec, kterým bude záznam končit. Nepříjemné ale je, že když zadáme takový oddělovač například jako XML značku `</doc>`, samotný oddělovač se již neobjeví na vstupu funkcí *map*, a tedy ani na výstupu. Přeprogramovat to není triviální. Tento přístup je v grafech označen Hadoop.

Druhý způsob je přepočítání zmíněných `splits` hned na začátku[1]. To nelze řešit paralelně, tedy se tím výpočet zdrží. V grafech je tento přístup označen jako Hadoop D. Používá ho například třída StreamXmlRecordReader. Ta ovšem není vhodná ke zpracování korpusů, protože nastavením jejích parametrů `begin=<doc> end=</doc>` docílíme zpracování korpusu tolika instancemi pro *map*, kolik je dokumentů, což velmi zpomaluje zpracování velkých korpusů s mnoha dokumenty. Proto tato práce představuje třídu REDelimInputFormat (viz příloha D), která bere jako parametr přibližnou velikost splitu a oddělovač. Hranice nastaví na prvním výskytu oddělovače za velikostí danou prvním parametrem. Tuto třídu používá Hadoop D. V grafu **??** je vidět, že jeho zpoždění za výpočtem Hadoop je úměrné velikosti dat, což odpovídá zpoždění na začátku při počítání splitů.

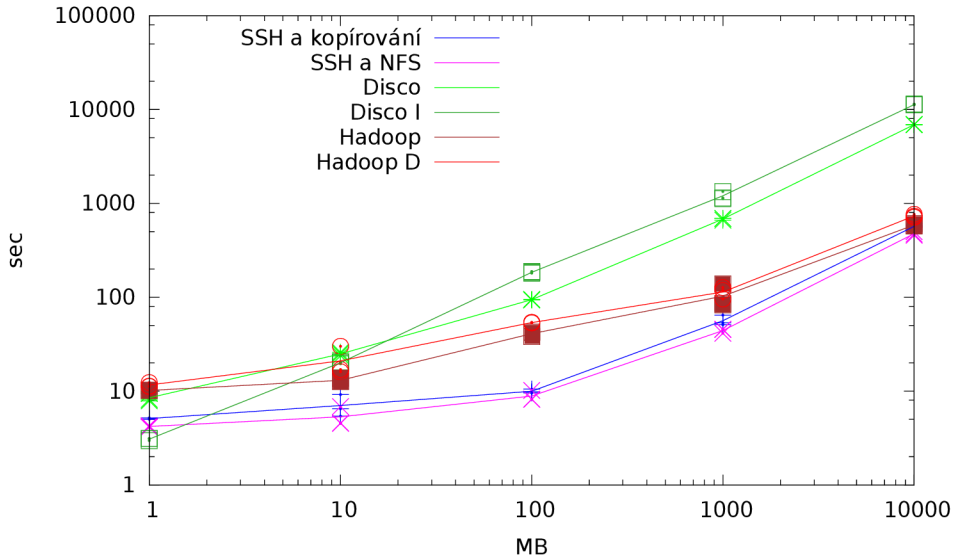
Zajímavé je ovšem i srovnání pro případ, že by data frameworků nebyla trvale uložena v DFS, ale vstup a výstup by se musel pokaždé kopírovat[2]. Takové srovnání ukazují grafy 7.5 (který opět vynechává extrémní hodnoty) a 7.6 s logaritmickou škálou. Jejich hodnoty jsou v tabulce C.3.

Grafy ukazují, že pro jednorázové použití, kdy se do DFS kopírují data jen kvůli jednomu výpočtu, se nehodí framework Disco. Dále je na nich vidět, že framework Hadoop sice jemně zaostává za jednoduchými přístupy, ale s velikostí dat se zmenšuje i význam tohoto rozdílu. Také ukazuje, že NFS je schopné zátěž zvládnout a jeví se jako nejlepší alternativa.

1. Split vskutku označuje hranice a neobsahuje data. Vlastně je to čtveřice $\langle cesta, zacatek, delka, uzly \rangle$.
2. Hodnoty pro „SSH a kopírování“ a „SSH a NFS“ tedy zůstávají stejné.



Obrázek 7.5: TreeTagger na strojích alba, aura, asteria04 a 62 nymfeXX i s nahráváním a stahováním dat.

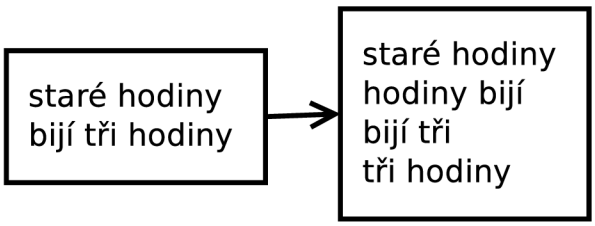


Obrázek 7.6: TreeTagger na strojích alba, aura, asteria04 a 62 nymfeXX i s nahráváním a stahováním dat.

7.3 Četnost bigramů

Výpočet četností n-gramů (sledu n po sobě jdoucích slov) je základní úlohou pro vytváření jazykových modelů používaných v NLP, například ve statistických přístupech k rozpoznávání řeči, strojovému překladu nebo syntaktické analýze. Postup úlohy je: rozdělení textu na slova, generování n-gramů slov, výpočet četností n-gramů.

Nejjednodušším případem úlohy je četnost unigramů, tedy samotných slov. Zde ovšem zcela odpadá fáze generování n-gramů, takže program pro výpočet unigramů se může zásadně lišit od programů pro výpočet ostatních n-gramů. Proto je pro účely práce jako jednoduchý příklad zvolena četnost bigramů (dvojic slov).



Obrázek 7.7: generování bigramů

Výpočet četností bigramů je úloha typu *reduce* a jako taková má několik parametrů, které mohou ovlivnit běh. Jedním z nich je použití nebo nepoužití funkcí *combiner* (C v popisku grafu). Tím hlavním ale je počet instancí provádějících *reduce* (číslo před R v popisku grafu). Defaultně nastavenou hodnotou bývá 1, ale na wiki systému Hadoop[3], je doporučení násobit počet tzv. workerů (jader vyhrazených pro běh úlohy) koeficienty 0.95 nebo 1.75, tedy konstantou o trochu menší než jedna nebo o trochu menší než dvě, proto, aby instance provádějící *reduce* pohodlně doběhly v jednom nebo ve dvou kolech a nezbyl žádný opozdilec.

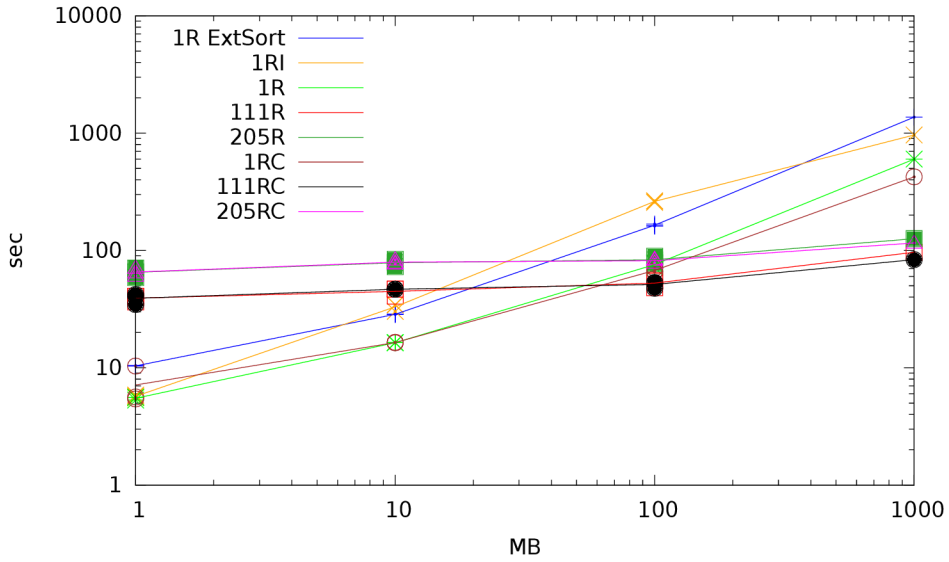
Výpočty jsou měřené na clusteru strojů alba, aura a asteria04, konfigurovaném na maximální kapacitu celkem 117 jader. Vynásobením výše zmíněnými koeficienty dostaneme čísla 111 a 205. Úlohy tedy budeme měřit pro 1, 111 a 205 instancí provádějících funkce *reduce*. Vždy s funkcemi *combiner* a bez nich.

7.3.1 Disco

Ve frameworku Disco existuje navíc možnost třídit externím příkazem sort (ExtSort v popisku grafu), což zpomaluje výpočet, ale je šetrné k paměti. Podobně je tu možnost číst data uložená v interním komprimovaném formátu, což také zpomaluje výpočet, jak jsme se přesvědčili v sekci 7.2. Pro tyto dvě – pro naše účely spíše zpátečnické – alternativy nejsou vyzkoušeny všechny možnosti, ale pro představu jsou v grafu 7.8 a v tabulce C.4 zařazeny každá jednou, s jednou instancí provádějící *reduce* a bez funkcí *combiner*.

Graf 7.8 ukazuje, že už pro data 100 MB se vyplatí nasadit 111 instancí provádějících *reduce* (pomocí koeficientu 0.95) lépe, než použít defaultní hodnotu 1. Ukazuje, že dvě kola funkcí *reduce* se na takto malých datech nevyplatí, a také je v něm vidět, že použití nebo nepoužití funkcí *combiner* nehraje moc velkou roli, ale ve všech třech případech platí, že na 1GB je vidět nepatrně lepší výsledek výpočtů, které funkce *combiner* použili.

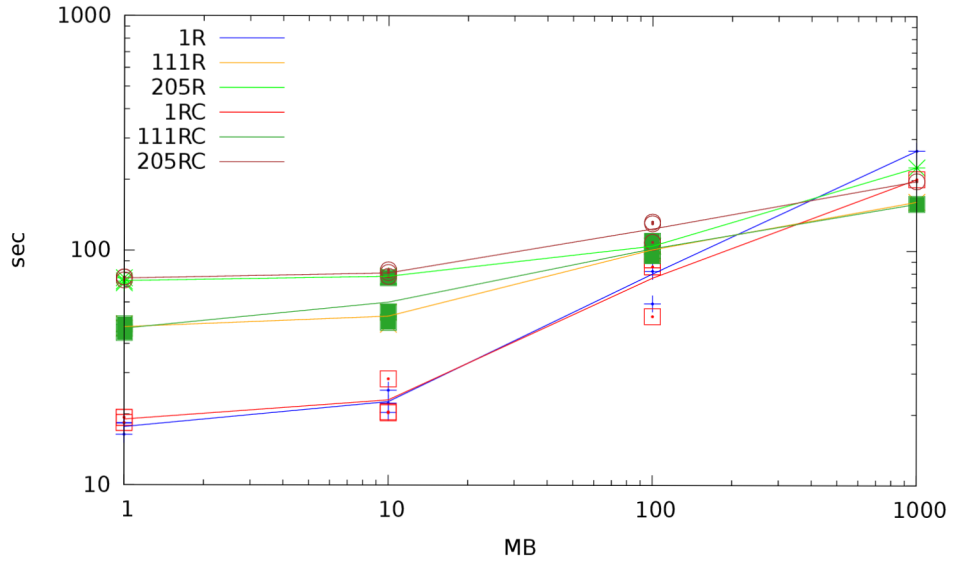
3. `http://wiki.apache.org/hadoop/HowManyMapsAndReduces`



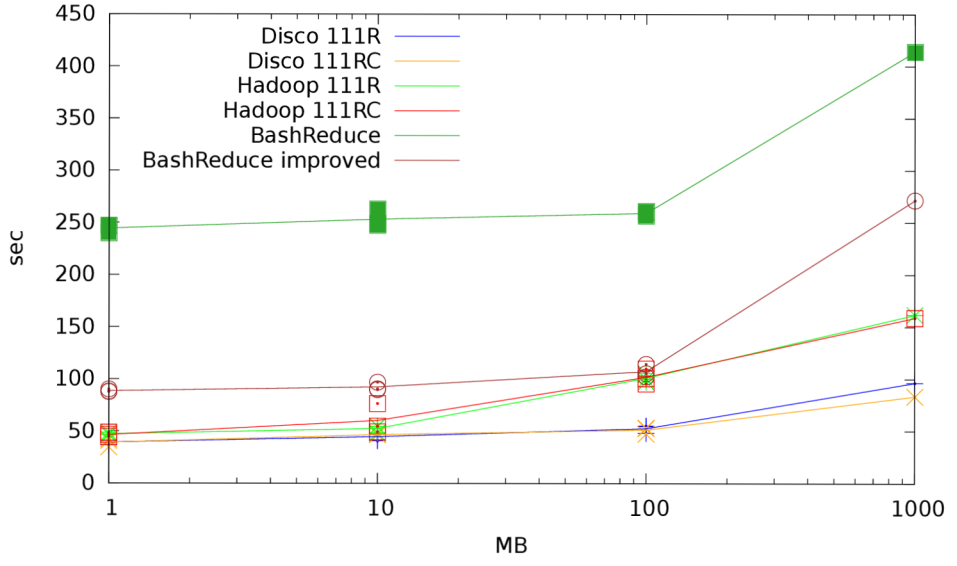
Obrázek 7.8: Výpočet četností bigramů ve frameworku Disco na strojích alba, aura a asteria04

Graf 7.9 ukazuje pro Hadoop přibližně to samé, co graf 7.8 pro Disco. Jenom nepatrnou výhodu nasazení funkcí *combiner* a nejvýhodnější počet reducerů 111 (tj. koeficient 0.95).

Závěrečný graf 7.10 ukazuje, že pro úlohy typu reduce dokáže být Disco přece jen lepší než Hadoop. Porovnává výpočty běžící se 111 instancemi provádějícími *reduce*. To je také nejlepší příležitost k porovnání se skriptem BashReduce, který nemá nastavitelný počet instancí pro *reduce*, ale spustí jich tolik, kolik má v konfiguraci jader clusteru. Je ovšem vidět, že kvůli postupnému navazování SSH spojení s každým uzlem, je Pomalý už na malých datech. Tato nevýhoda je částečně odstraněna ve verzi BashReduce improved, která vznikla v rámci této práce, a která se snaží přihlásit spuštěním mnoha procesů programu SSH paralelně. Je ovšem vidět, že už pro 1 GB obě verze ztrácejí na frameworky MapReduce, protože nemají distribuovaný souborový systém.



Obrázek 7.9: Výpočet četností bigramů ve frameworku Hadoop na strojích alba, aura a asteria04



Obrázek 7.10: Shrnutí výpočtů četností bigramů frameworky Disco a Hadoop se 111 instancemi pro *reduce* a jejich porovnání se skriptem BashReduce na strojích alba, aura a asteria04.

Kapitola 8

Závěr

Výsledky pro značkování korpusů TreeTaggerem ukazují, že už pro data 10GB se vyplatí jejich ukládání do nějakého distribuovaného souborového systému. Z druhé strany ukazují, že pokud je preferované kopírovat do DFS vstupní data před každou úlohou, nehodí se nasadit framework Disco. Framework Hadoop má obstojné výsledky, ale pro úlohy typu map, jako je tato není lepší než jednoduché přístupy k paralelizaci. Posledním zajímavým výsledkem pro značkování TreeTaggerem je, že zde byl velice úspěšný přístup "SSH a NFS" přesto, že výpočet běžel na 65 uzlech (dohromady na 365 jádrech).

Příloha A

WordCount.java

```
1.  package org.myorg;
2.
3.  import java.io.IOException;
4.  import java.util.*;
5.
6.  import org.apache.hadoop.fs.Path;
7.  import org.apache.hadoop.conf.*;
8.  import org.apache.hadoop.io.*;
9.  import org.apache.hadoop.mapred.*;
10.     import org.apache.hadoop.util.*;
11.
12.     public class WordCount {
13.
14.        public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, T
15.          private final static IntWritable one = new IntWritable(1);
16.          private Text word = new Text();
17.
18.          public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> ou
19.            String line = value.toString();
20.            StringTokenizer tokenizer = new StringTokenizer(line);
21.            while (tokenizer.hasMoreTokens()) {
22.              word.set(tokenizer.nextToken());
23.              output.collect(word, one);
24.            }
25.          }
26.        }
27.
28.        public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable
29.          public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, In
30.            int sum = 0;
31.            while (values.hasNext()) {
32.              sum += values.next().get();
33.            }
34.            output.collect(key, new IntWritable(sum));
35.          }
36.        }
37.
38.        public static void main(String[] args) throws Exception {
39.          JobConf conf = new JobConf(WordCount.class);
40.          conf.setJobName("wordcount");
41.
42.          conf.setOutputKeyClass(Text.class);
43.          conf.setOutputValueClass(IntWritable.class);
44.
45.          conf.setMapperClass(Map.class);
46.          conf.setCombinerClass(Reduce.class);
47.          conf.setReducerClass(Reduce.class);
```

```
48.
49.             conf.setInputFormat(TextInputFormat.class);
50.             conf.setOutputFormat(TextOutputFormat.class);
51.
52.             FileInputFormat.setInputPaths(conf, new Path(args[0]));
53.             FileOutputFormat.setOutputPath(conf, new Path(args[1]));
54.
55.             JobClient.runJob(conf);
57.         }
58.     }
```

(převzato z `http://hadoop.apache.org/docs/r1.0.4/mapred_tutorial.html#Example%3A+Wo`

Příloha B

split_offsets.py

`split_offsets.py` vytiskne bajtové pozice na kterých lze rozdělit soubor podle zadaných kritérií. Umožňuje specifikovat oddělovač `DELIM_RE`, na kterém může rozdělení nastat, dále přibližnou velikost nebo počet dílů.

```
Usage: ./split_offsets.py [OPTION] FILE1 [FILE2 ...]
      OPTIONS:
      -b SIZE        put SIZE bytes per chunk
      -d DELIM_RE    expands offset to the first regular expression DELIM_RE
      -h             print help and exit
      -n CHUNKS      generate CHUNKS offsets

      DELIM_RE is a single line regular expression matching every start of new chunk. e.g. '<[dD
      SIZE is an integer and optional unit (example: 10M is 10*1024*1024).  Units are K, M, G, T
      (powers of 1024) or KB, MB, ... (powers of 1000). Default size is 64MB.
      SIZE may be l/N (e.g. l/20MB). It expands size to the first start of the new line.
      CHUNKS may be: N       print N offsets
                     l/N     print N offsets without splitting lines
```

```
#!/usr/bin/python
import getopt
import math
import sys
import os
import re

def print_help() :
    sys.stderr.write("Usage: " + sys.argv[0] + """ [OPTION] FILE1 [FILE2 ...]
      OPTIONS:
      -b SIZE        put SIZE bytes per chunk
      -d DELIM_RE    expands offset to the first regular expression DELIM_RE
      -h             print help and exit
      -n CHUNKS      generate CHUNKS offsets

      DELIM_RE is a single line regular expression matching every start of new chunk. e.g. '<[dD
      SIZE is an integer and optional unit (example: 10M is 10*1024*1024).  Units are K, M, G, T
      (powers of 1024) or KB, MB, ... (powers of 1000). Default size is 64MB.
      SIZE may be l/N (e.g. l/20MB). It expands size to the first start of the new line.
      CHUNKS may be: N       split into N files based on size of input
                     l/N     split into N files without splitting lines
""")

def find_re_or_nl(f, re_obj, split_lines, start, end) :
    """
    @param file    f    opened_file
    @param _sre.SRE_Pattern or None    re_obj  compiled single line regular expression
    @param int  start   search from the offset start
```

```
    @param int  end   search to the offset end
    @param bool split_lines if false, offset of line beginning is returned

    @return int  offset of first re_obj occurrence; -1 if no occurence; offset of the closest ne
    """
    if re_obj:
        f.seek(start, os.SEEK_SET)
        offset = start
        for line in f :
            found = re_obj.search(line)
            if found and not split_lines:
                return offset
            elif found and offset + found.start(0) < end:
                return offset + found.start(0)
            else:
                offset += len(line)
            if offset >= end :
                break
    else:
        if start == 0:
            return 0
        else:
            start -= 1;
            f.seek(start, os.SEEK_SET)
            offset = start + len(f.readline())
            if offset < end:
                return offset
    return -1

def parse_units(s):
    """
    @param  string   s~number and unit (e.g. 200kb) matching  "\d+[KMGTPEZY]B?" case insensitive
    @return int bytes
    """
    units = ['K', 'M', 'G', 'T', 'P', 'E', 'Z', 'Y' ]
    power = 1024
    num, unit = re.match("(\d+)([KMGTPEZY]B?)?",s,re.I).groups('B')
    if unit in [ 'b', 'B' ]:
        return int(num)
    elif unit[-1] in ['b', 'B']:
        power=1000
    exp = units.index(unit[0].upper()) + 1
    return int(num) * power ** exp

def print_offsets(bytes_, files, delim_re, split_lines):
    sum_of_sizes = 0
    idx = 0
    offset = bytes_
    for f_name in files:
        f_size = os.path.getsize(f_name)
        f = open(f_name, 'r')
        while offset < f_size:
            idx = find_re_or_nl(f, delim_re, split_lines, offset, offset + bytes_)
            if idx > -1:
                print sum_of_sizes + idx
                sys.stdout.flush()
            offset += bytes_
        f.close()
        offset -= f_size
```

```
        sum_of_sizes += f_size


############## PROCESS ARGUMENTS ##############
delim_re = None
chunks = None
bytes_ = None
split_lines = True
optlist, files = getopt.getopt(sys.argv[1:], "b:d:hn:")
for opt in optlist:
    if opt[0] == '-b' and opt[1].startswith('l/'):
        split_lines = False
        bytes_ = parse_units(opt[1][2:])
    elif opt[0] == '-b':
        bytes_ = parse_units(opt[1])
    elif opt[0] == '-d':
        if opt[1] != "":
            delim_re = re.compile(opt[1])
    elif opt[0] == '-h':
        print_help()
        sys.exit(0)
    elif opt[0] == '-n' and opt[1].startswith('l/'):
        split_lines = False
        chunks = int(opt[1][2:])
    elif opt[0] == '-n':
        chunks = int(opt[1])
    else:
        print_help()
        sys.exit(2)

if len(files) < 1:
    print_help()
    sys.exit(2)

if chunks is None and bytes_ is None:
    bytes_ = 64000000

if chunks is not None and bytes_ is not None:
    print >>sys.stderr, sys.argv[0]+': cannot split in more than one way'
    sys.exit(1)

###################### MAIN ####################
length = 0
for f_name in files:
    length += os.path.getsize(f_name)
if not bytes_:
    bytes_ = int(math.ceil(length / float(chunks)))
elif not chunks:
    chunks = int(math.ceil(length / float(bytes_)))

if not delim_re and split_lines:
    i = 1
    #while i < chunks: # for i in range(1, chunks) fell down with OverFlowError
    while bytes_ * i < length:
        print bytes_ * i
        sys.stdout.flush()
        i += 1
else:
    print_offsets(bytes_, files, delim_re, split_lines)
```

```
print length
```

Příloha C

Naměřené hodnoty

MB	BDFS	DDFS push	DDFS chunk	DDFS to_doc_reader	HDFS
1	0.822	3.756	0.903	0.450	1.167
1	0.804	3.497	0.892	0.454	1.157
1	0.762	4.706	0.787	0.534	1.180
10	1.042	12.280	5.807	2.189	1.292
10	1.028	10.884	5.866	2.196	1.388
10	1.019	11.739	6.162	2.166	1.336
100	1.621	33.666	55.243	19.866	2.957
100	1.501	34.231	58.642	19.436	2.822
100	1.612	37.388	56.972	20.456	2.607
1000	10.646	210.343	521.698	196.894	17.150
1000	10.035	214.192	506.916	194.407	16.255
1000	9.973	220.943	521.655	206.426	15.557
10000	92.400	1629.218	5936.494	2088.175	140.705
10000	91.688	1648.589	5737.730	1924.261	127.122
10000	95.223	1592.367	5721.772	1935.279	120.814

Tabulka C.1: Nahrávání dat ze stroje alba do DFS na stroje alba, aura, asteria04 a 62 nymfeXX.

Každý řádek představuje jiná data vybraná z korpusu EnClueWeb. Uvedená je jejich přiblžná velikost v milionech bajtů a trvání nahrávání v sekundách.

MB	SSH a kopírování	SSH a NFS	Disco	Disco I	Hadoop	Hadoop D
1	5.184	4.118	2.870	1.494	e	8.990
1	10.985	4.277	2.838	1.496	7.778	8.915
1	5.190	4.214	2.874	1.530	7.845	9.934
10	5.410	4.526	5.214	8.648	9.852	e
10	6.495	4.574	4.489	9.120	10.836	13.184
10	9.175	6.850	4.734	8.780	9.769	14.113
100	9.547	8.210	7.415	82.944	35.889	47.540
100	10.494	8.220	6.731	79.358	35.910	47.420
100	9.846	10.168	6.792	76.690	31.947	47.420
1000	64.437	45.250	17.342	166.647	46.110	77.510
1000	50.950	41.420	18.732	217.653	44.936	89.445
1000	53.320	45.589	21.560	e	e	58.766
10000	475.854	499.150	181.184	391.714	192.380	349.400
10000	491.485	457.651	177.298	389.577	178.392	293.414
10000	463.332	471.850	182.258	400.210	186.434	322.777

Tabulka C.2: TreeTagger na strojích alba, aura, asteria04 a 62 nymfeXX.

Každý řádek představuje jiná data vybraná z korpusu EnClueWeb. Uvedená je jejich přiblžná velikost v milionech bajtů a trvání výpočtů v sekundách.

MB	SSH a kopírování	SSH a NFS	Disco	Disco I	Hadoop	Hadoop D
1	5.184	4.118	8.154	3.132	1.167	11.224
1	e	4.277	7.878	3.121	10.123	11.273
1	5.019	4.214	9.434	2.976	10.215	12.323
10	5.401	4.526	25.935	19.549	12.681	30.058
10	6.495	4.574	23.81	20.242	13.732	16.085
10	9.175	6.85	24.954	20.455	12.635	16.989
100	9.547	8.201	93.67	187.912	42.49	54.1
100	10.494	8.202	93.987	183.866	42.278	53.603
100	9.846	10.168	94.95	180.579	38.138	53.448
1000	64.437	45.25	691.327	1126.6	85.924	118.133
1000	50.95	41.402	662.736	1143	82.911	126.86
1000	53.302	45.589	690.684	1339.68	138.916	95.724
10000	568.254	499.105	6920.4	11437.6	612.84	765.944
10000	583.173	457.651	6936.04	11237.3	580.865	712.351
10000	558.565	471.805	6884.62	11232	575.208	729.358

Tabulka C.3: TreeTagger na strojích alba, aura, asteria04 a 62 nymfeXX i s nahráváním a stahováním dat.

Každý řádek představuje jiná data vybraná z korpusu EnClueWeb. Uvedená je jejich přiblžná velikost v milionech bajtů a trvání výpočtů v sekundách.

MB	1R ExtSort	1RI	1R	111R	205R	1RC	111RC	205RC
1	10.335	5.730	5.588	40.586	70.874	5.636	42.130	70.854
1	10.389	5.844	5.308	36.460	58.823	5.415	40.657	64.842
1	10.425	5.547	5.595	40.620	65.572	10.359	34.559	60.695
10	28.580	30.269	16.376	46.768	78.879	16.354	46.710	77.012
10	28.541	34.414	16.352	46.726	72.739	16.358	46.725	78.221
10	28.599	34.431	16.346	40.756	84.156	16.480	46.708	83.349
100	167.728	264.847	78.677	54.668	83.118	65.446	52.842	84.102
100	161.976	263.361	72.048	55.181	89.196	70.810	53.363	83.150
100	161.659	257.180	76.785	48.225	77.151	66.252	47.543	78.188
1000	1367.517	964.024	600.834	96.406	126.048	424.065	83.413	115.814

Tabulka C.4: Výpočet četností bigramů frameworkem Disco na strojích alba, aura a asteria04.

Každý řádek představuje jiná data vybraná z korpusu EnClueWeb. Uvedená je jejich přiblžná velikost v milionech bajtů a trvání výpočtů v sekundách.
Ve sloupcích je před R počet instancí vykonávajících *reduce*. Písmeno C symbolizuje použití funkcí *combiner*.

MB	1R	111R	205R	1RC	111RC	205RC
1	18.373	46.469	73.711	18.404	46.436	76.744
1	16.402	48.507	72.507	19.360	44.492	74.533
1	18.312	46.449	76.035	19.338	48.446	76.582
10	25.358	54.466	77.638	20.509	54.683	80.495
10	20.395	48.452	77.493	20.321	49.408	77.499
10	22.319	54.532	77.650	28.356	76.511	82.615
100	97.752	99.910	104.505	85.503	100.591	130.404
100	59.438	105.645	104.608	91.615	109.932	108.666
100	81.713	97.562	104.827	52.429	95.578	132.665
1000	267.018	161.758	226.895	201.818	158.762	197.872

Tabulka C.5: Výpočet četností bigramů frameworkem Hadoop na strojích alba, aura a asteria04.

Každý řádek představuje jiná data vybraná z korpusu EnClueWeb. Uvedená je jejich přiblžná velikost v milionech bajtů a trvání výpočtů v sekundách.
Ve sloupcích je před R počet instancí vykonávajících *reduce*. Písmeno C symbolizuje použití funkcí *combiner*.

MB	Disco 111R	Disco 111RC	Hadoop 111R	Hadoop 111RC	BashReduce	BashReduce improve
1	40.586	42.130	46.469	46.436	246.568	89.98
1	36.460	40.657	48.507	44.492	246.829	87.62
1	40.620	34.559	46.449	48.446	239.826	87.92
10	46.768	46.710	54.466	54.683	247.406	96.91
10	46.726	46.725	48.452	49.408	263.003	90.45
10	40.756	46.708	54.532	76.511	249.256	89.80
100	54.668	52.842	99.910	100.591	261.027	105.47
100	55.181	53.363	105.645	109.932	259.943	114.46
100	48.225	47.543	97.562	95.578	256.554	102.26
1000	96.406	83.413	161.758	158.762	414.071	271.49

Tabulka C.6: Shrnutí výpočtů četností bigramů frameworky Disco a Hadoop se 111 instancemi pro *reduc* jejich porovnání se skriptem BashReduce na strojích alba, aura a asteria04.

Každý řádek představuje jiná data vybraná z korpusu EnClueWeb. Uvedená je jejich přiblžná velikost v m onech bajtů a trvání výpočtů v sekundách.

Ve sloupcích je před R počet instancí vykonávajících *reduce*. Písmeno C symbolizuje použití funkcí *combin*

Příloha D

REDelimInputFormat

```
package redelim;

import java.io.IOException;
import java.util.ArrayList;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.compress.CompressionCodecFactory;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileSplit;
import org.apache.hadoop.mapred.InputSplit;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.JobConfigurable;
import org.apache.hadoop.mapred.LineRecordReader;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.util.LineReader;


public class REDelimInputFormat extends FileInputFormat<LongWritable, Text>
        implements JobConfigurable {

    private CompressionCodecFactory compressionCodecs = null;

    @Override
    public void configure ( JobConf conf ) {
      compressionCodecs = new CompressionCodecFactory( conf );
    }

    @Override
    protected boolean isSplitable ( FileSystem fs, Path file ) {
      return compressionCodecs.getCodec( file ) == null;
    }

    @Override
    public InputSplit[] getSplits ( JobConf job, int numSplits )
                                                throws IOException {
      String delimRegEx = job.get("mapred.input.format.delim");
      String ss =  job.get( "mapred.input.format.split.size" );
      Long splitSize;
      if( ss == null || ss.length() == 0 ){
          splitSize = (long) 64 * 1024 * 1024;
```

```
    } else {
        splitSize = Long.valueOf( ss );
    }

    Pattern delim = Pattern.compile( delimRegEx );

    ArrayList<FileSplit> splits = new ArrayList<FileSplit>();
    for ( FileStatus status : listStatus( job ) ) {
      Path fileName = status.getPath();
      if (status.isDir()) {
        throw new IOException("Not a file: " + fileName);
      }
      FileSystem  fs = fileName.getFileSystem( job );
      LineReader lr = null;
      Matcher delim_matcher;
      try {
        FSDataInputStream in  = fs.open( fileName );
        lr = new LineReader( in, job );
        Text line = new Text();
        long begin = 0;
        long length = 0;
        int lineLen;
        while ( ( lineLen = lr.readLine( line ) ) > 0 ) {
          length += lineLen;
          if ( length >= splitSize && (delim_matcher = delim.matcher( line.toString() )).find(
            length = length - lineLen + delim_matcher.start();
            splits.add( new FileSplit( fileName, begin, length, new String[]{} ) );
            begin += length;
            length = lineLen - delim_matcher.start();
          }
        }
        if ( length != 0 ) {
          splits.add( new FileSplit( fileName, begin, length, new String[]{} ) );
        }

      } finally {
        if ( lr != null ) {
          lr.close();
        }
      }
    }
    return splits.toArray(new FileSplit[splits.size()]);
  }

  @Override
  public RecordReader<LongWritable, Text>
          getRecordReader( InputSplit split,
                           JobConf job,
                           Reporter reporter ) throws IOException {
    reporter.setStatus(split.toString());
    return new LineRecordReader(job, (FileSplit) split);
  }
}
```

Literatura

[1] Disco distributed filesystem. http://disco.readthedocs.org/en/0.4.5/howto/ddfs.html.

[2] Zookeeper. http://wiki.apache.org/hadoop/ZooKeeper.

[3] D Borthakur. The hadoop distributed file system: Architecture and design, 2007. hadoop.apache.org.

[4] Sanjay Ghemawat Jeffrey Dean. mapreduce: Simplified data processing on large clusters. *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[5] Helmut Schmid. Probabilistic part-of-speech tagging using decision trees. *Proceedings of International Conference on New Methods in Language Processing*, 1994.

[6] Tom White. *Hadoop: The Definitive Guide, 2nd Edition*. 2011.