

NÁSKOK
DÍKY
ZNALOSTEM

PROFINIT

B0M33BDT

Technologie pro velká data

MapReduce

Milan Kratochvíl, Sergii Stamenov

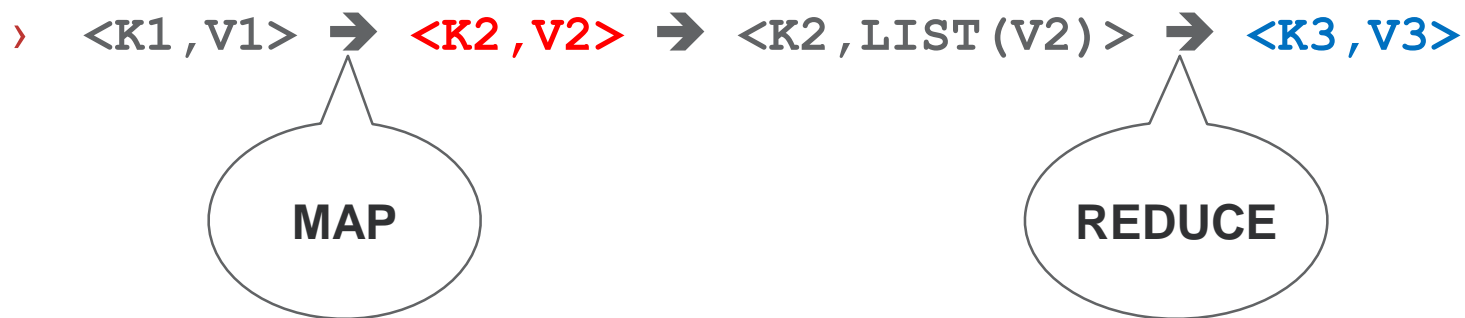
6. 11. 2019

MapReduce



MapReduce

- › Paradigma/framework/programovací model pro distribuované paralelní výpočty
- › Princip:
 - MAP
 - vstupní data ve formátu $\langle \text{KEY1}, \text{VALUE1} \rangle$ konvertuje na $\langle \text{KEY2}, \text{VALUE2} \rangle$
 - REDUCE
 - vstupní data ve formátu $\langle \text{KEY2}, \text{LIST}(\text{VALUE2}) \rangle$ konvertuje na $\langle \text{KEY3}, \text{VALUE3} \rangle$



MapReduce – příklad: Počet slov podle délky

MAP

**SHUFFLE
& SORT**

REDUCE

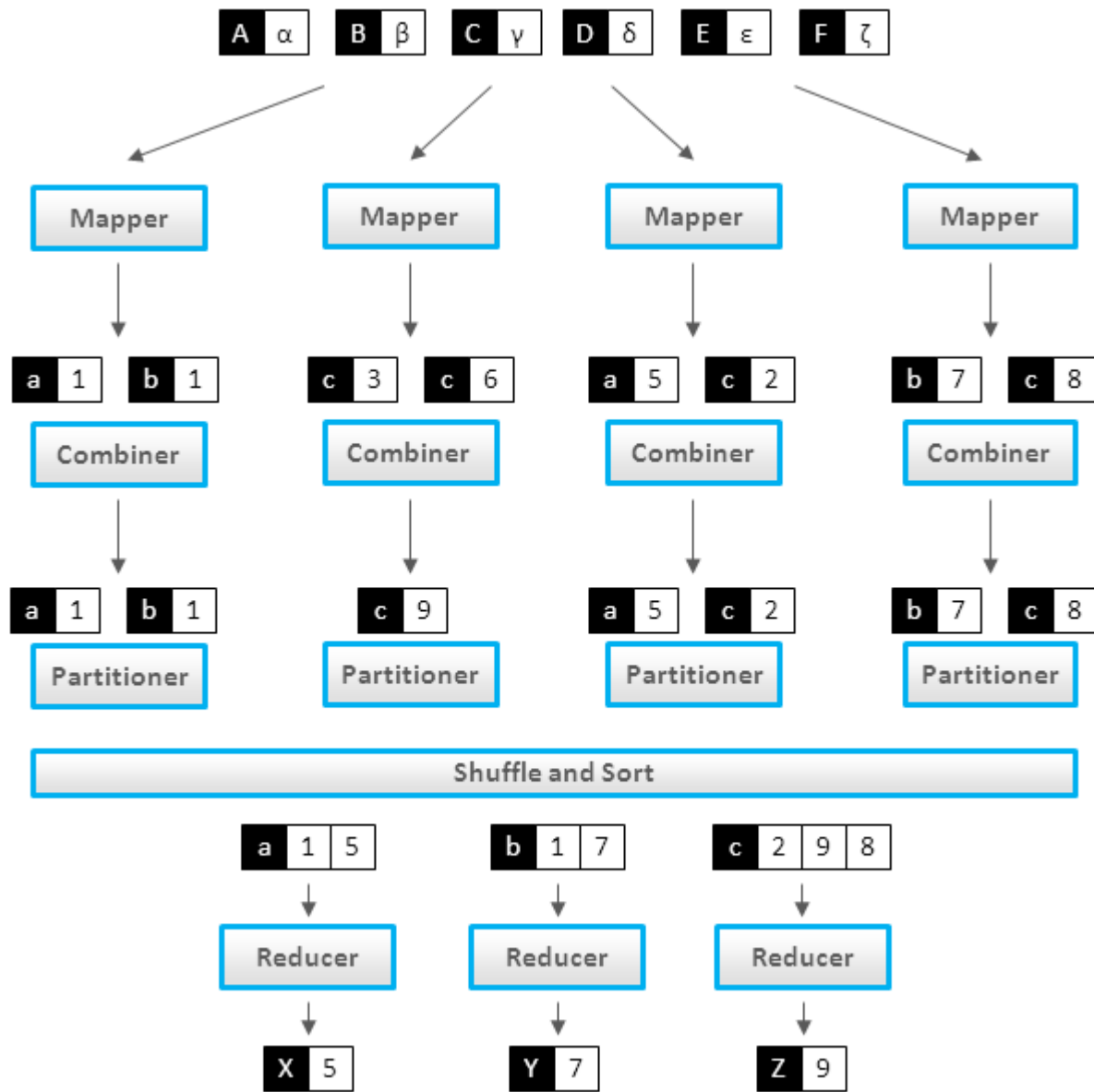
Ó, náhlý déšť již zvířil prach a čilá laň teď
běží s houfcem gazel k úkrytům.

1: ó
5: náhlý
4: déšť
1: a
...

1: LIST(ó, a, s, k)
3: LIST(již, laň, teď)
4: LIST(déšť, čilá, běží)
5: LIST(náhlý, prach, gazel)
...

1: 4
3: 3
4: 3
5: 3
6: 1
7: 2

MapReduce podrobně



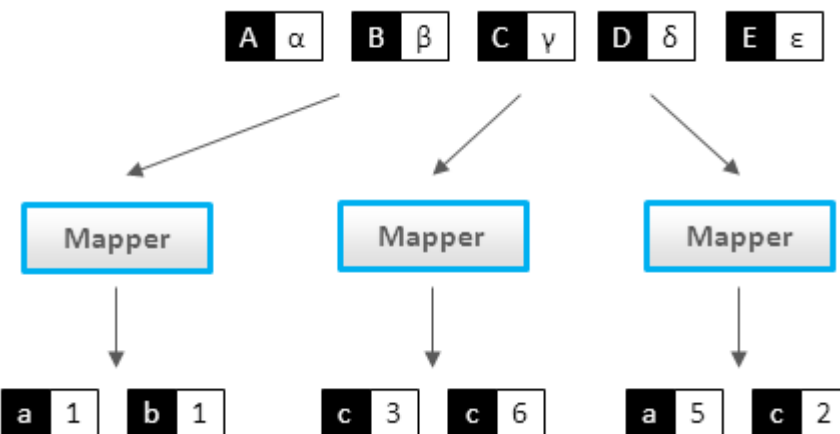
Příprava

- › Definice zdrojových dat
- › **Split**
 - Pokud je soubor větší než 1 HDFS blok, splitne se na menší
- › Podle rozmístění jednotlivých bloků naplánuje YARN optimálně, které nody spustí které joby
 - Snaha o maximální lokálnost
 - Snaha o eliminaci „zbytečných“ transferů přes síť
 - To všechno usnadňuje replikace dat v HDFS
- › Pokud to ale povaha problému neumožňuje, lze zakázat splittování
 - pak každý mapper zpracuje celý soubor.

Mapper

$\langle K1, V1 \rangle \rightarrow \langle K2, V2 \rangle$

- › Vstupní údaje **mapuje** na hodnoty typu klíč – hodnota
- › Vstupem je formálně také klíč – hodnota
 - Ale často nás klíč vůbec nezajímá! Je to třeba offset souboru
- › Klíč se může libovolně opakovat, hodnota může být různá
- › Data jsou na konci seříděna podle klíče (zpravidla, ne vždy)
- › Typicky probíhá v mnoha paralelních jobech
 - Každý soubor, resp. split je zpracován samostatným mapperem
- › **Výstup se zapisuje na lokální disk**



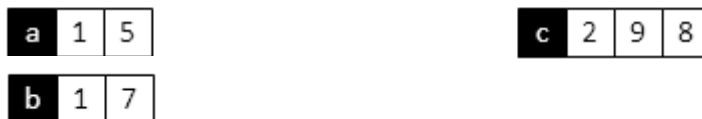
Ó, náhlý déšť již zvířil prach a čilá laň teď běží s houfcem gazel k úkrytům.

1: ó
1: a
1: k
1: s
3: laň
3: teď
3: již
4: déšť
5: běží
...

Shuffle & Sort

$\langle K2, V2 \rangle \rightarrow \langle K2, LIST(V2) \rangle$

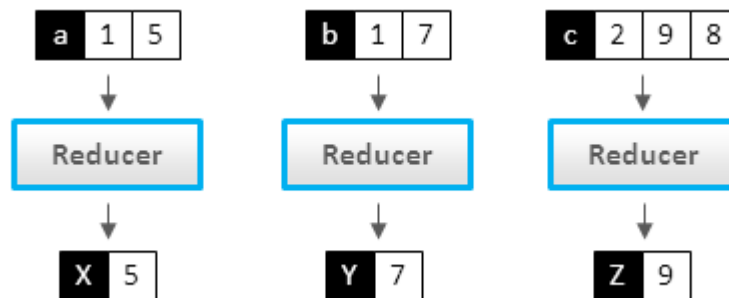
- › Probíhá během/po skončení mapperů
- › Vstupem jsou **vygenerované soubory z mapperů**
- › Všechna data se **stejným klíčem slije na jeden node**
 - Tj. musí data načíst z jednotlivých nodů (disková operace)
 - Ale čte je jen z lokálních disků, nikoli z HDFS
 - A pak musí data přenést (sít')
- › Setřídí data podle klíče (merge)
- › Optimalizace – malá data pošle rovnou do reduceru, velká merguje na lokálním disku
- › **Typicky nejnáročnější operace**



Reduce

$\langle K2, LIST(V2) \rangle \rightarrow \langle K3, V3 \rangle$

- › Čte produkovaná pomocí Shuffle & Sort
- › „Redukuje“ list hodnot čtených z výstupy Shuffle & Sort
- › Zpravidla je reducerů (řádově) méně než mapperů
- › Počet lze definovat, default je 1
- › Každý reducer generuje 1 soubor do HDFS
- › Výstup se nijak nesortuje



Combiner

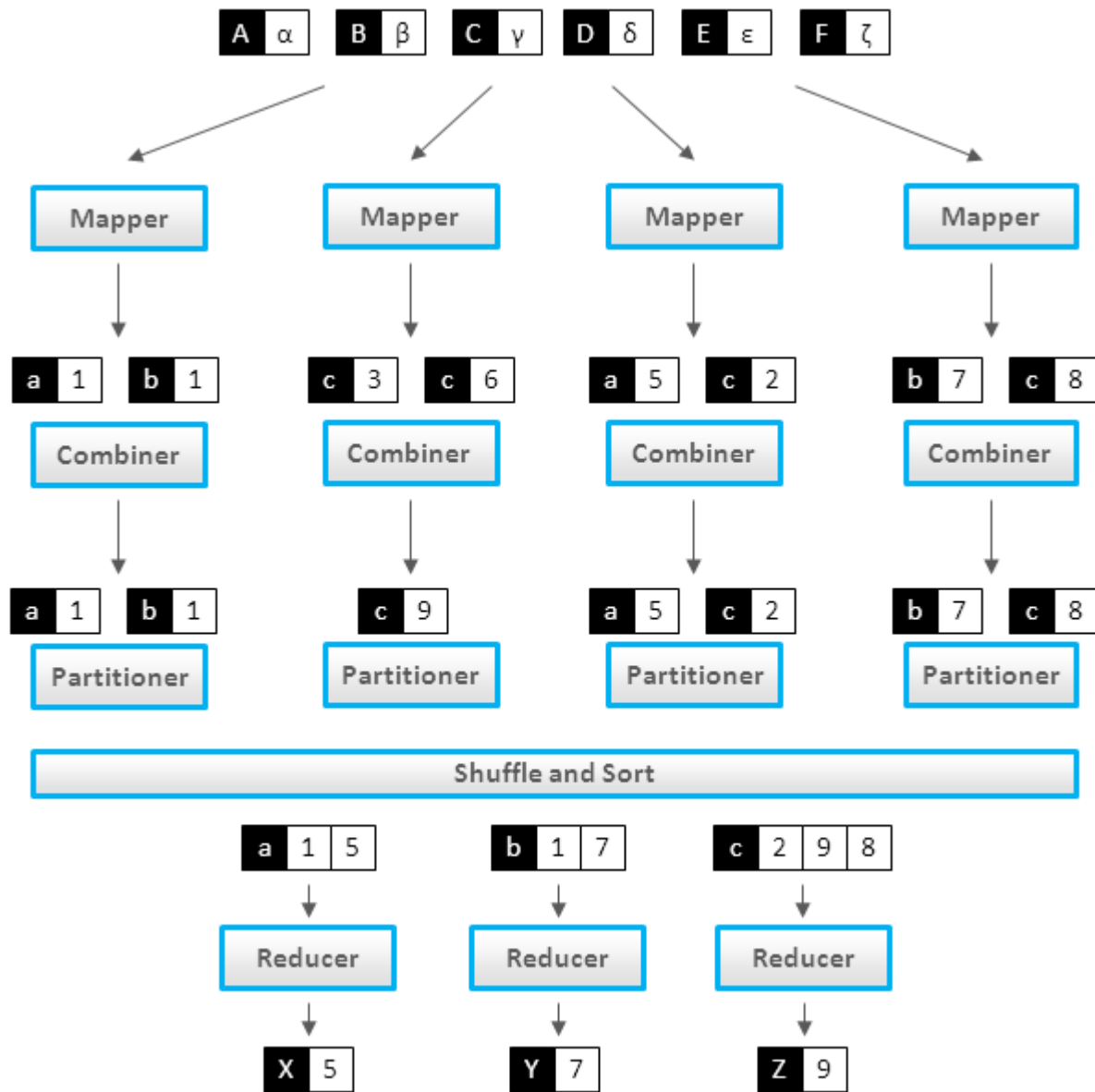
$\langle K2, LIST(V2) \rangle \rightarrow \langle K3, V3 \rangle$

- › Volitelná část zpracování mapperu
- › Provádí reduce na straně mapperu
- › Stejné rozhraní jako Reducer
- › Proč použít combiner
 - data jsou načtena v paměti, ušetří se IO operace – čtení
 - často je výstup menší než vstup, ušetří se IO operace – zápis (a následné čtení)
- › Nemá zpravidla smysl, pokud combiner generuje více dat než je jejich vstup
- › Často lze použít stejný kód jako pro reducer!
 - Anebo si lze napsat úplně vlastní

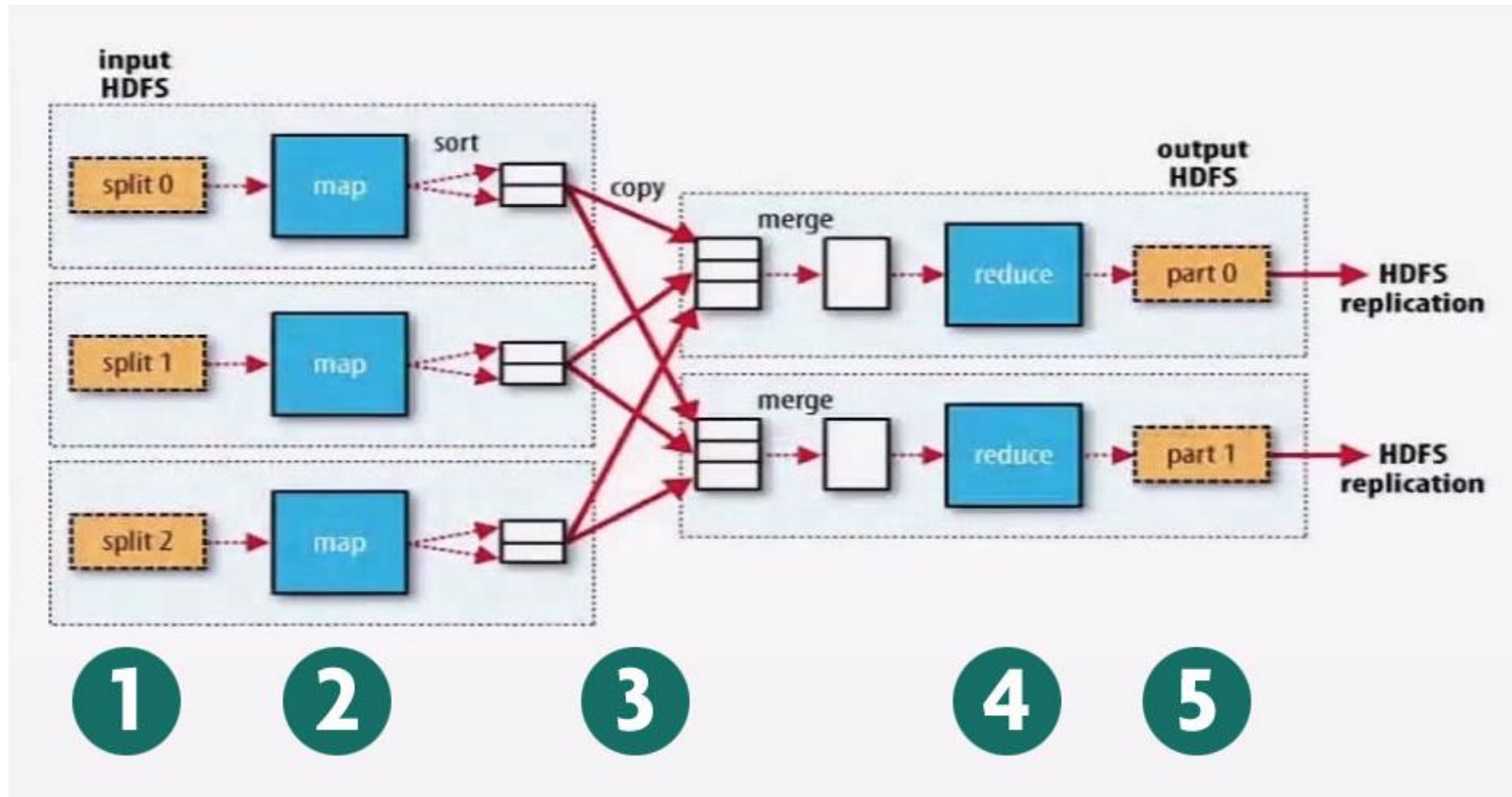
Partitioner

- › Partitioner rozděluje data do několika partitions
 - podle klíče
 - podle hodnoty
- › MapReduce zajišťuje, že jedna partition je zpracována jedním reducerem
- › Defaultně – počítá se `hash klíče modulo počet reducerů`
 - to pro většinu případů znamená velmi rovnoměrné rozdělení do jednotlivých partitions
- › Vlastní partitioner lze použít např. v případech, kdy
 - chci znát nějaké specifické rozdělení (např. věkové kategorie)
 - mám velmi nevyvážené klíče (jeden klíč se vyskytuje abnormálně často) – problém celebrit

MapReduce podrobně



MapReduce toky dat



<https://content.pivotal.io/blog/hadoop-101-programming-mapreduce-with-native-libraries-hive-pig-and-cascading>

Třída úloh, které lze MapReduce zpracovat

- › Metoda „Rozděl a panuj“
- › **Formální požadavky**
- › Každá MapReduce operace \blacktriangle musí splňovat
 - a) asociativitu, tj. $(A \blacktriangle B) \blacktriangle C = A \blacktriangle (B \blacktriangle C)$
 - b) existenci neutrálního prvku \diamond , tj. $A \blacktriangle \diamond = A$
 - c) komutativita, tj. $(A \blacktriangle B) = (B \blacktriangle A)$ [přísně vzato není povinná]
- › Proč
 - a) neovlivníme pořadí vykonávaných operací
 - b) node, který nemá výstup/nezpracovává data, neovlivní výsledek
 - c) mapper/shuffle může změnit pořadí dat
komutativita umožňuje použít combiner

Příklady – vhodné úlohy

- › „Školní úlohy“
 - počet slov v textu
 - četnost slov
- › Praktičtější úlohy
 - reporting – načítání řady dílčích výsledků (prodeje)
 - podle klienta
 - produktu
 - lokality
 - řazení dat (sortování)
 - výpis pořadí např. prodávaných výrobků podle prodaných kusů
 - filtrování dat
 - validace
 - hledání „distinktních“ hodnot (SELECT DISTINCT a FROM t)
 - extrakce snímků z videa

Příklady – nevhodné úlohy

- › Medián
 - nesplňuje asociativitu
- › Průměr
 - nesplňuje asociativitu
- › Násobení matic
 - nesplňuje komutativitu
- › ALE: Úlohy lze vhodně přeformulovat!

Průměr

10 15 96	Split 1
33 57 21 746	Split 2



- › Combiner
 - spočítat průměr
- › Výstup mapperu
 - SPLIT1:40.3333
 - SPLIT2:214.25
- › Výstup reduceru
 - AVG:234.4166666666



- › Combiner
 - spočítat průměr **a počet**
- › Výstup mapperu
 - SPLIT1:40.3333:3
 - SPLIT2:214.25:4
- › Výstup reduceru
 - AVG:139.714

Java API

Java API – příklad

- › Zajímá nás četnost výskytu jednotlivých slov v textu

Java API - Mapper

```
1 package eu.profinnit.hadoop;
2
3 import java.io.IOException;
4
5 import org.apache.hadoop.io.LongWritable;
6 import org.apache.hadoop.io.Text;
7 import org.apache.hadoop.mapreduce.Mapper;
8 import org.apache.hadoop.mapreduce.lib.input.FileSplit;
9
10 public class MRMapper extends Mapper < LongWritable, Text, Text, LongWritable > {
11
12     @Override
13     protected void map(LongWritable key, Text value,
14         Mapper < LongWritable, Text, Text, LongWritable > .Context context)
15         throws IOException,
16         InterruptedException {
17
18         String[] words = value.toString().split("[\\.,;!\\?\\|\\(\\)\\s]");
19
20         for (String word: words) {
21             if (word.length() > 4) {
22                 context.write(new Text(word.toUpperCase()), new LongWritable(1));
23             }
24         }
25     }
26 }
27
28 }
```

Java API – Reducer

```
1 package eu.profinnit.hadoop;
2
3 import java.io.IOException;
4
5 import org.apache.hadoop.io.LongWritable;
6 import org.apache.hadoop.io.Text;
7 import org.apache.hadoop.mapreduce.Reducer;
8
9 public class MRReducer extends Reducer < Text, LongWritable, Text, LongWritable > {
10
11     @Override
12     protected void reduce(Text key, Iterable < LongWritable > values,
13         Reducer < Text, LongWritable, Text, LongWritable > .Context context)
14         throws IOException,
15         InterruptedException {
16
17         long sum = 0;
18
19         for (LongWritable count: values) {
20             sum += count.get();
21         }
22
23         context.write(key, new LongWritable(sum));
24     }
25
26 }
```

Java API – Driver

```
1 package eu.profnit.hadoop;
2
3 import org.apache.hadoop.conf.Configuration;
4 import org.apache.hadoop.fs.Path;
5 import org.apache.hadoop.io.LongWritable;
6 import org.apache.hadoop.io.Text;
7 import org.apache.hadoop.mapreduce.Job;
8 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
9 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
10
11 public class MRDriver {
12
13     public static void main(String[] args) throws Exception {
14
15         Configuration conf = new Configuration();
16         conf.set("mapreduce.output.textoutputformat.separator", ";");
17
18         Job job = Job.getInstance(conf);
19
20         job.setJarByClass(MRDriver.class);
21
22         job.setJobName("CetnostSlov");
23         job.setMapperClass(MRMapper.class);
24         job.setReducerClass(MRReducer.class);
25         job.setOutputKeyClass(Text.class);
26         job.setOutputValueClass(LongWritable.class);
27         job.setMapOutputKeyClass(Text.class);
28         job.setMapOutputValueClass(LongWritable.class);
29
30         FileInputFormat.addInputPath(job, new Path(args[0]));
31         FileOutputFormat.setOutputPath(job, new Path(args[1]));
32
33         job.setNumReduceTasks(4);
34
35         boolean success = job.waitForCompletion(true);
36
37         System.exit(success ? 0 : 1);
38     }
39 }
```

Java API – Combiner

- › Lze provést optimalizaci velmi snadným způsobem:
 - `job.setCombinerClass(MRReducer.class);`
- › Combiner je použit stejný jako Reducer!
 - Ale pozor, toto neplatí vždy!
- › Výrazně se omezí množství přenášených dat

Java API – další možnosti

- › Map a Reduce probíhají záznam po záznamu
- › Je možné také spustit kód při inicializaci a ukončení celého procesu
- › Podobné pro Mapper i Reducer
- › `setup()`
 - spustí se jedenkrát při inicializaci jobu
 - typicky si zde načtu data z cache
- › `cleanup()`
 - typicky uvolnění externích zdrojů (zavření souborů atd.)

Jak pustit MapReduce program?

1. Zkompilovat zdrojové kódy do JAR
2. Spustit příkaz

```
hadoop jar wc.jar eu.profnit.hadoop.MRDriver /data/capek /data/vystup
```

3. Výstup

```
17/03/21 14:50:46 INFO mapred.LocalJobRunner: Starting task:
attempt_local699809073_0001_m_000003_0
17/03/21 14:50:46 INFO output.FileOutputCommitter: File Output Committer Algorithm version is 1
17/03/21 14:50:46 INFO mapred.Task: Using ResourceCalculatorProcessTree : [ ]
17/03/21 14:50:46 INFO mapred.MapTask: Processing split: hdfs://localhost/data/capek/Karel_Capek-
-Povidky_z_druhe_kapsy.txt:0+274512
17/03/21 14:50:46 INFO mapred.MapTask: (EQUATOR) 0 kvi 26214396(104857584)
17/03/21 14:50:46 INFO mapred.MapTask: mapreduce.task.io.sort.mb: 100
17/03/21 14:50:46 INFO mapred.MapTask: soft limit at 83886080
17/03/21 14:50:46 INFO mapred.MapTask: bufstart = 0; bufvoid = 104857600
17/03/21 14:50:46 INFO mapred.MapTask: kvstart = 26214396; length = 6553600
17/03/21 14:50:46 INFO mapred.MapTask: Map output collector class =
org.apache.hadoop.mapred.MapTask$MapOutputBuffer
17/03/21 14:50:46 INFO mapred.LocalJobRunner:
17/03/21 14:50:46 INFO mapred.MapTask: Starting flush of map output
17/03/21 14:50:46 INFO mapred.MapTask: Spilling map output
17/03/21 14:50:46 INFO mapred.MapTask: bufstart = 0; bufend = 327529; bufvoid = 104857600
17/03/21 14:50:46 INFO mapred.MapTask: kvstart = 26214396(104857584); kvend =
26135552(104542208); length = 78845/6553600
17/03/21 14:50:46 INFO mapreduce.Job: map 30% reduce 0%
17/03/21 14:50:46 INFO mapred.MapTask: Finished spill 0
17/03/21 14:50:46 INFO mapred.Task: Task:attempt_local699809073_0001_m_000003_0 is done. And is
in the process of committing
17/03/21 14:50:46 INFO mapred.LocalJobRunner: map
17/03/21 14:50:46 INFO mapred.Task: Task 'attempt_local699809073_0001_m_000003_0' done.
```

Co je výstupem programu?

```
[cloudera@quickstart capek_new]$ hadoop fs -ls /data/vystup
```

```
Found 5 items
```

```
-rw-r--r--      3 cloudera supergroup          0 2017-03-21 14:50 /data/vystup/_SUCCESS
-rw-r--r--      3 cloudera supergroup    181885 2017-03-21 14:50 /data/vystup/part-r-00000
-rw-r--r--      3 cloudera supergroup    177191 2017-03-21 14:50 /data/vystup/part-r-00001
-rw-r--r--      3 cloudera supergroup    176376 2017-03-21 14:50 /data/vystup/part-r-00002
-rw-r--r--      3 cloudera supergroup    179476 2017-03-21 14:50 /data/vystup/part-r-00003
```

```
[cloudera@quickstart capek_new]$ hadoop fs -cat /data/vystup/part-r-00002 | head -n 20
```

```
*TROMS;1
1876-77;1
80-86425-58-4;1
ABECEDU;1
ABELIÍ;1
ABOVARIO;;1
ABOVO;1
ABSOLUTISM;5
ABSOLUTISMEM;1
ABSOLUTISTICKÝ;1
ABSOLUTISTIČTÍ;1
ABSOLUTISTŮ;1
ABSOLUTNOSTI;1
ABSOLUTNÍM;1
ABSOLVOVAT;1
ABSTRAKTNÍCH;2
ABSURDUM;1
ABYCHOM;72
ACERJAPONICUM;1
ACETYLSALICYLAZID;1
```

Příklady

„SQL“

- › Hive jako tradiční dotazovací nástroj používá MapReduce. Jak?
- › `SELECT COUNT(*) FROM SOME_TABLE;`
- › **MAP**
 - pro každý řádek zapíšeme stejný klíč a číslo 1
 - `__row | 1`
- › **REDUCE**
 - jeden jeden klíč!
 - na výstup zapíšeme počet (nebo sumu) hodnot ke klíči row
 - `__count | 633214`

„SQL“

```
› SELECT COUNT(*), NAME FROM SOME_TABLE  
WHERE COUNTRY='CZ'  
GROUP BY NAME  
HAVING COUNT(*) > 100;
```

› MAP

- přeskočíme všechny záznamy, které nemají CZ
- pro každý řádek zapíšeme stejný klíč – NAME
- hodnota bude 1
- Novak|1
Sladek|1

› REDUCE

- tolik klíčů, kolik máme jmen!
- spočítáme ke každému klíči počet (nebo sumu) hodnot ke klíči
- na výstup zapíšeme, jen ty, které jsou větší než 100
- Novak|654
Sladek|162

Joinování

- ›

```
SELECT S.SALES, S.YEAR, C.NAME
FROM ALL_SALES S JOIN ALL_CUSTOMER C
ON S.CUST_ID = C.CUST_ID
```
- › Map-side join
 - mapování probíhá na straně mapperu
 - zpravidla vyžaduje připravit data
 - není třeba přenášet mnoho dat do reduceru
- › Reduce-side join
 - jednodušší na zápis
 - ale méně efektivní
- › MapReduce driver
 - je třeba spustit 2 mappery pro každou tabulku
 - ```
MultipleInputs.addInputPath(job, salesInputPath,
 TextInputFormat.class, SalesMapper.class);
```

```
MultipleInputs.addInputPath(job, custInputPath,
 TextInputFormat.class, CustMapper.class);
```

# Reduce-side join

- › MAP - CustMapper
  - na výstupu je klíč joinovací atribut (CUST\_ID)
  - hodnota na výstupu je identifikace tabulky a NAME
  - `ID447:CustMapper:Novak Jaroslav`
- › MAP – SalesMapper
  - na výstupu je klíč joinovací atribut (CUST\_ID)
  - hodnota na výstupu je identifikace tabulky a SALES a YEAR
  - `ID447:SalesMapper:254122:2016`
- › REDUCE
  - Iterace přes prvky values
  - Pro každý klíč dostanu (patrně) jeden záznam z CustMapper a několik ze SalesMapper
  - `ID447:SalesMapper:254122:2016:Novak Jaroslav`



# Map-side join

- › Typicky druhý soubor **je výrazně menší a vejde se do paměti**
- › Použití DistributedCache
- › MAP
  - v metodě `setup()` načtení hodnot z DistributedCache do paměti (třeba HashMap pro rychlý přístup)
  - v samotné metodě `map()` načítám data z většího souboru a dohledávám k němu údaje v lokální cache
- › REDUCE
  - nic zvláštního

# Co je MapReduce algoritmus?

- › Řada složitých algoritmů si nevystačí se zpracováním v jednom běhu MapReduce.
- › Typické je řetězení
  - jeden program vytvoří/připraví výstup do dočasného úložiště
  - následující program načte data z dočasného úložiště jako vstup
- › Není nutné řetězit celé MapReduce programy, je možné řetězit jen několik mapperů pracujícími nad týmiž daty
  - pracují ve stejném JVM – odpadá režie s inicializací VM
  - typicky filtry, validace

## Příklad na řetězení

- › 

```
SELECT S.SALES, S.YEAR, C.NAME
FROM ALL_SALES S JOIN ALL_CUSTOMER C
ON S.CUST_ID = C.CUST_ID
ORDER BY NAME;
```
  
- › Předpokládám použití Reduce-side join
- › 1. MapReduce provede joinování
- › 2. MapReduce
  - klíč na výstupu mapperu bude NAME
  - dojde k sortování podle klíče
  - v reduceru pak změním pořadí údajů na původní, ale dostaneme správně seřazený seznam

# MapReduce – shrnutí

# Vlastnosti MapReduce

- › Odladěná implementace
- › Spolehlivé
  - když ostatní nástroje „padají“, MapReduce jede
- › Velmi „malé“ požadavky na operační paměť
  - v paměti se drží jen malé bloky
  - slučování fragmentů probíhá merge sortem na discích
- › Velmi četné zapisování na disk
  - nicméně na HDFS je jenom finální vstup a výstup
- › Často generuje násobky dočasných dat ve srovnáním z množství vstupních dat
  - a často většinu dat musí zároveň uložit
- › Toto je základní rys, zároveň silná i slabá stránka
- › Tedy ve srovnání s jinými nástroji výrazně pomalejší

## Slabé stránky

- › Dlouhá doba, než se zpracování spustí
  - desítky sekund až minuty
- › Při startu se inicializuje nový proces, JVM, nahrávají se knihovny...
  - a to se děje v mnoha instancích (na druhou stranu to separuje případné problémy do izolovaného procesu)
- › Někdy složité na implementaci (mnoho knihoven)
- › Mnoho konfigurací

# Kdy použít MapReduce

- › Tam, kde úloha umožňuje paralelizaci
  - a lze ji převést na úlohu vhodnou pro MapReduce
- › Při práci se skutečně velkými daty (petabajty)
  - tak velká data se v paměti nedají uchovávat/zpracovat
- › Při práci s výpočetně náročnými úlohami, kde se spíše data načítají než zapisují
- › Když není čas kritický
- › Stačí dávkové zpracování
- › Když není dostatek paměti v clusteru

# Kdy není vhodný MapReduce

- › Tam, kde úlohu nelze paralelizovat a převést na MapReduce kompatibilní 😊
- › Je-li vyžadována okamžitá reakce
  - i pro jednoduché úlohy trvá inicializace velmi dlouho
- › Když je úkol „malý“



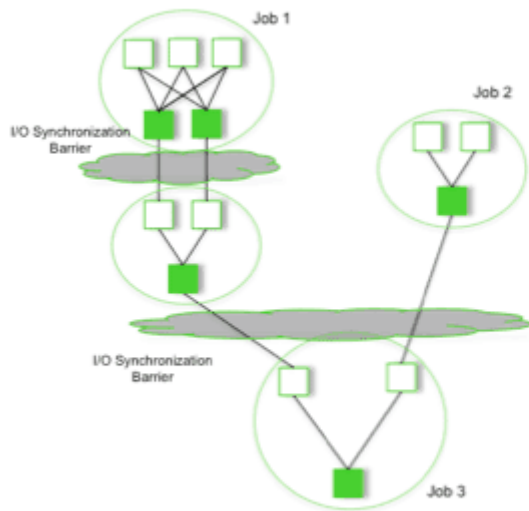
# Kde se používá

- › Kdekoli
- › Zpracování velkého množství strukturovaných dat
  - ETL
  - **HIVE**
- › Text mining
- › Zpracování obrazu a videa
- › Zkoumání lidského genomu
- › Statistické modely
- › Některé algoritmy strojového učení
  - např. K-Means

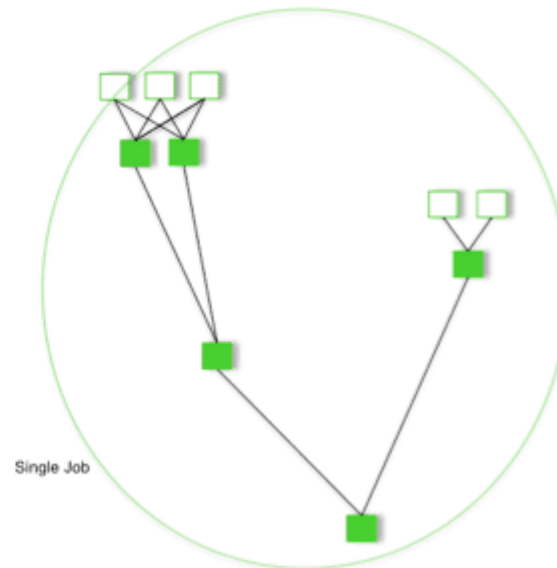
# Alternativy v Hadoop

## › Apache Tez

- optimalizovaná „verze“ MapReduce
- princip je podobný, stále se zapisují mezivýsledky na disk
- ale množství zápisů se optimalizuje, některé úkoly se slučují
- násobně zvyšuje rychlost a prostupnost dat



Pig/Hive - MR

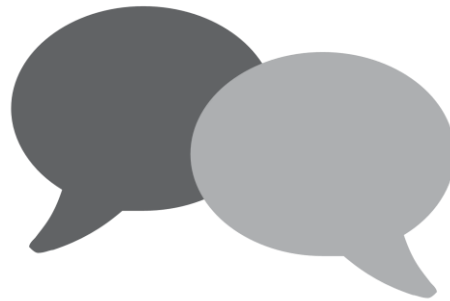


Pig/Hive - Tez

# Alternativy v Hadoop

- › Apache Spark
  - příští přednáška
  - podobné paradigma
  - místo ukládání mezivýsledků na disk vše drží v paměti
  - často řádové rozdíly ve výkonu
  - vhodný i pro ad hoc analýzy
  
- › Přijďte příště a dozvíte se více!

# Diskuze



`milan.kratochvil@profininit.eu`

# Díky za pozornost

**PROFINIT**

NÁSKOK DÍKY ZNALOSTEM

Profinit EU, s.r.o.

Tychonova 2, 160 00 Praha 6 | Telefon + 420 224 316 016



Web  
[www.profinit.eu](http://www.profinit.eu)



LinkedIn  
[linkedin.com/company/profinit](https://linkedin.com/company/profinit)



Twitter  
[twitter.com/Profinit\\_EU](https://twitter.com/Profinit_EU)



Facebook  
[facebook.com/Profinit.EU](https://facebook.com/Profinit.EU)



Youtube  
Profinit EU