

OMO

8 - MapReduce v NoSQL světě

- Principy MapReduce
- Motivační příklady
- Vybrané patterny: Sumarizační, filtrační, organizační, join patterny

Ing. David Kadleček, PhD

kadlecd@fel.cvut.cz, david.kadlecek@cz.ibm.com

Formulace problému

Úloha: Máme historická data o počasí mezi lety 2000 a 2015. Tato data obsahují průměrné teploty za jednotlivé dny a jednotlivé oblasti. Jsou uloženy v místech, kde se tato data sbírají. Chceme najít nejvyšší teplotu v každém roce z dat přes celou zeměkouli.

!

- **Problém kritické cesty:** Zpoždění stroje při řešení pod-úlohy by nemělo zpoždit následující milník nebo celou úlohu.
- **Spolehlivost jednotlivých částí systému:** Co kdyby vypadla některá komponenta systému
- **Rovnoměrné rozdělení úlohy:** Jak rozdělit úlohu na menší bloky tak, aby komplexita a časová náročnost byla odpovídající výkonnosti stroje => žádná samostatný stroj by neměl být přetížen nebo nevyužíván.
- **Jedna podčást úlohy se nespočítá:** Pokud některý ze strojů nespočítá výstup, tak nebudeme schopni vypočítat finální výsledek. Musí existovat mechanismus, který je schopný sestavit řešení i v této situaci.
- **Agregace výsledku:** Měl by existovat mechanismus pro agregaci výsledku, který generuje každý stroj do finálního výstupu.

=> **MapReduce framework** řeší tyto výzvy

=> Nutnou podmínkou je řešitelnost úlohy principem **Rozděl a panuj**

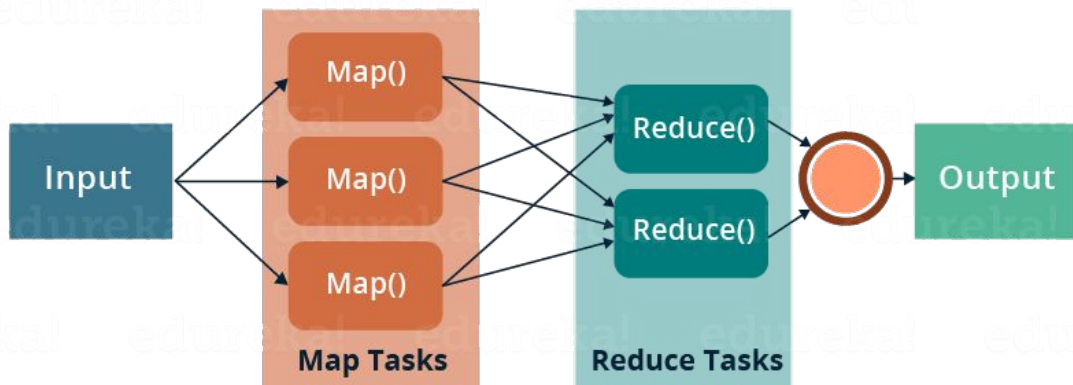
=> Relaxuji řešení tak, že se spokojím i se **suboptimálním výsledkem**

MapReduce framework

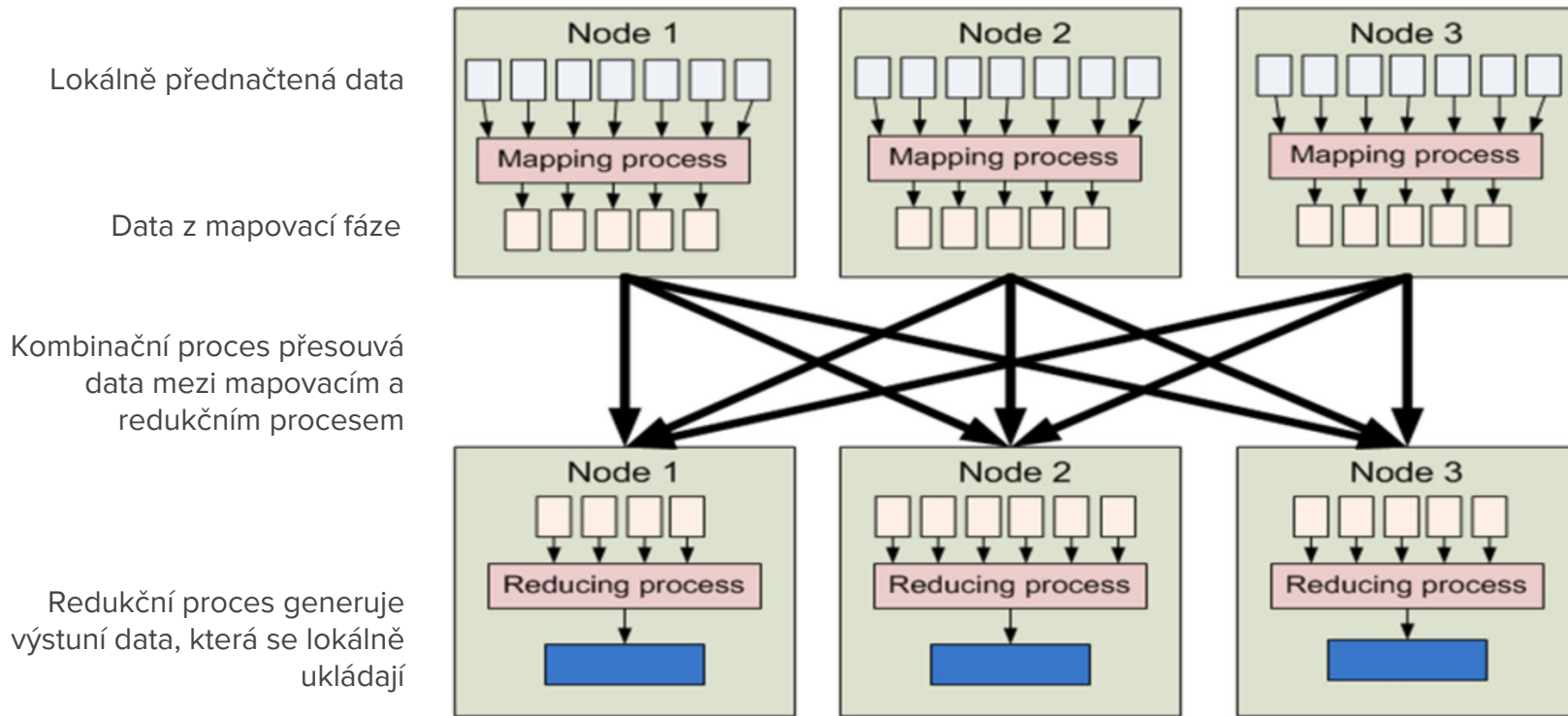
MapReduce je programovací framework, který umožňuje řešit úlohy, ve kterých se paralelně počítají úlohy nad obrovským množstvím dat, která se nacházejí v různých lokalitách. Pro řešitele je fyzické uložení dat, jejich distribuovanost, stejně jako distribuovanost výpočetního výkonu plně transparentní.

Funguje v následující sekvenci:

1. Input - načtení a preprocessing dat
2. **Map** - přemapování vstupních dat do key-value párů
3. Shuffle - kombinování (přeskládání) mezivýstupů pro reduce fázi
4. **Reduce** - redukce ~ agregace dat do menších key-value párů
5. Output - postprocessing a zápis dat



MapReduce framework

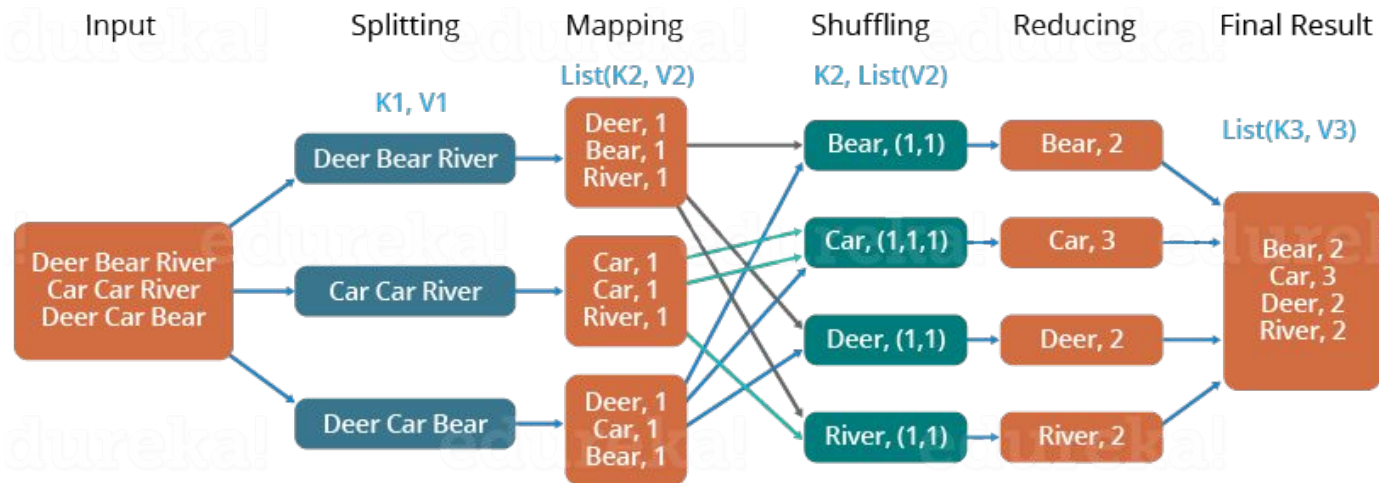


MapReduce framework - příklad

Máme velké množství souborů uložených na různých strojích v různých lokalitách. Pro zjednodušení uvažujme jeden soubor *example.txt*, který má následující obsah: *Deer, Bear, River, Car, Car, River, Deer, Car, Bear*

Cílem je spočítat výskyty jednotlivých slov a ty zapsat na výstup

The Overall MapReduce Word Count Process



MapReduce framework - příklad

1. Nejprve rozdělíme vstup na tři kusy (může odpovídat i tomu, že máme soubory v různých lokalitách). Toto dělení rozkládá práci na více strojů.
2. Poté každý kus projdeme a přepíšeme do mapy key-value párů, kde klíč je slovo a value je hodnota 1 (výskyt)
3. Dále se data přeskládají tak, aby každý reducer zpracovával stejné slovo... a pošlou do reduceru.
4. Po fázi mapování a míchání bude mít každý reducer jedinečný klíč a seznam hodnot odpovídajících tomuto klíči. Například Bear, [1,1]; Auto, [1,1,1], atd.
5. Každý reducer nyní počítá hodnoty, které jsou v tomto seznamu hodnot přítomny. Reducer dostane seznam hodnot, které jsou pro klíč Bear [1,1]. Poté počítá počet těch v samotném seznamu a dává konečný výstup jako - Bear, 2.
6. Na závěr jsou všechny výstupní páry key-value posbírány a zapsány do výstupního souboru.

MapReduce framework - příklad

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop...
public class WordCount {
    public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
        public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                value.set(tokenizer.nextToken());
                context.write(value, new IntWritable(1));
            }
        }
    }
    public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
            int sum = 0;
            for (IntWritable x : values) {
                sum += x.get();
            }
            context.write(key, new IntWritable(sum));
        }
    }
}
```

MapReduce framework - příklad

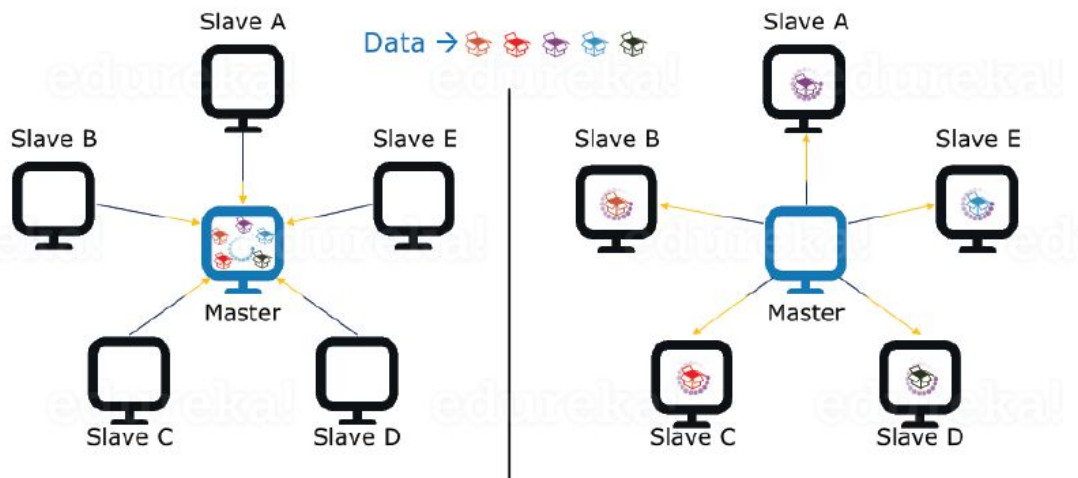
Při konfiguraci MapReduce jobu se nastaví třídy, které zpracovávají jednotlivé fáze MapReduce.

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = new Job(conf, "My Word Count Program");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    Path outputPath = new Path(args[1]);
    //Configuring the input/output path from the filesystem into the job
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    //deleting the output path automatically from hdfs so that we don't have to delete it explicitly
    outputPath.getFileSystem(conf).delete(outputPath);
    //exiting the job only if the flag value becomes false
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```


Výhody MapReduce

Paralelní zpracování - úloha je rozdělena na základě principu Rozděl a panuj a je řešena paralelně

Lokální zpracování - místo toho, abychom data přesouvali do centra, kde probíhá proces zpracování, tak přesouváme proces zpracování tam, kde jsou data



Výhody MapReduce

Mapreduce stojí na CAP teorému: pro distribuovaný systém není možné poskytovat více jak dvě záruky z těchto tří:

- **Consistency** - všechny klientské programy, které komunikují s distribuovaným systémem dostávají stejná data
- **Availability** - data lze získat z distribuovaného systému i v případě, že došlo k výpadku několika nodů
- **Partition tolerance** - distribuovaný systém funguje i když nelze komunikovat mezi některými nody

Versus ACID, na kterém stojí tradiční systémy jako např. Relační databáze:

- **Atomicita** - Databázová transakce je jako operace dále nedělitelná (atomární). Provede se buď jako celek, nebo se neprovede vůbec (a daný databázový systém to dá uživateli na vědomí, např. chybovým hlášením).
- **Konzistence** - Při a po provedení transakce není porušeno žádné integritní omezení.
- **Izolovanost** - Operace uvnitř transakce jsou skryty před vnějšími operacemi. Vrácením transakce (operací ROLLBACK) není zasažena jiná transakce, a když ano, i tato musí být vrácena.
- **Trvalost** - Změny, které se provedou jako výsledek úspěšných transakcí, jsou skutečně uloženy v databázi, a již nemohou být ztraceny.

Sumarizační patterny - agregátor

Cíl: Zgrupování záznamů dohromady podle klíčového atributu a spočítání agregační funkce nad takto zgrupovanými záznamy. Pokud θ je obecná agregační funkce, tak jí aplikujeme na seznam hodnot ($v_1, v_2, v_3, \dots, v_n$) abychom získali λ , kde $\lambda = \theta(v_1, v_2, v_3, \dots, v_n)$. Příklady agregační funkce θ jsou *minimum*, *maximum*, *průměr*, *medián*, *standardní odchylka* a další.

Motivace: Většina dat se kterými se člověk setkává je příliš jemná a v přílišném množství, aby z nich mohl člověk vyzorovat něco zajímavého.

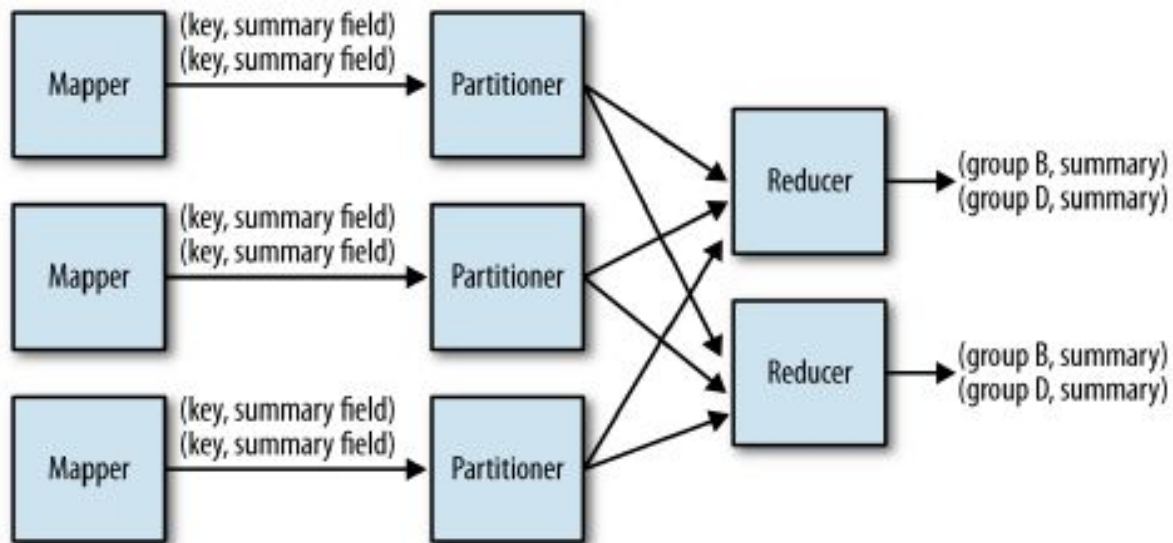
Webová stránka loguje aktivity uživatele (login, uživatelské dotazy, další akce), výsledkem čehož jsou terabyty dat v log souborech.

- ⇒ zgrupování aktivit podle hodiny a počtu záznamů nám umožní vytvořit histogram ze kterého vidíme kdy je naše webová stránka aktivní
- ⇒ Zgrupování reklam podle typu ve spojení s aktivitou nám pomůže určit jak efektivní jsou naše reklamní bannery

Snažíme se o něco podobného jako je SQL:

```
SELECT MIN(numericalcol1), MAX(numericalcol1), COUNT(*) FROM table GROUP BY groupcol2;
```

Sumarizační patterny – agregátor



Mapovací fáze: klíče obsahují hodnoty atributů podle kterých chceme grupovat a číslo, které budeme agregovat

Kombinační fáze: Sloučíme záznamy, které mají stejné klíče a ty pošleme do redukce

Redukční fáze: Reducer dostane skupinu a k ní seznam čísel které agreguje

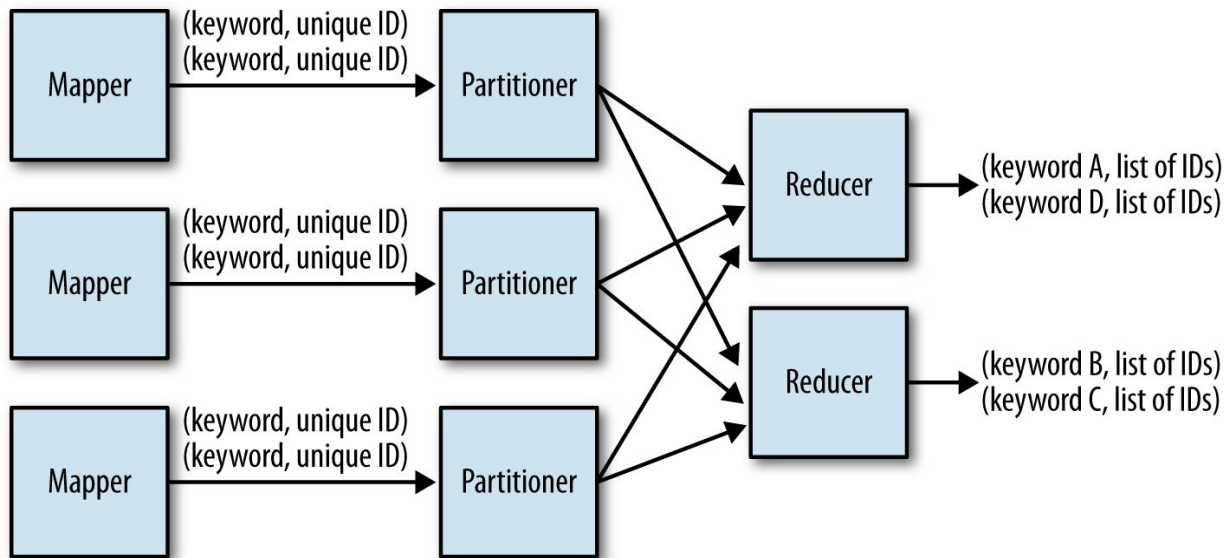
Inverzní index

Cíl: Vytvořit index k datům, abychom mohli provádět rychlejší hledání.

Motivace: Chceme vyhledávat ve webových stránkách pomocí klíčových slov. Procházet všechny stránky po zadání klíčového slova je nepoužitelné, protože trvá extrémně dlouho.

⇒ Projdeme všechny webové stránky dopředu a vytvoříme si mapu klíčové slovo – seznam webových stránek ve kterých se klíčové slovo vyskytuje. Konstrukce indexu sice trvá dlouho a zabírá místo, ale po jeho vytvoření lze každý dotaz provést extrémně rychle pouze lookupem z mapy.

Sumarizační patterny – inverzní index



Filtrační patterny – filtr

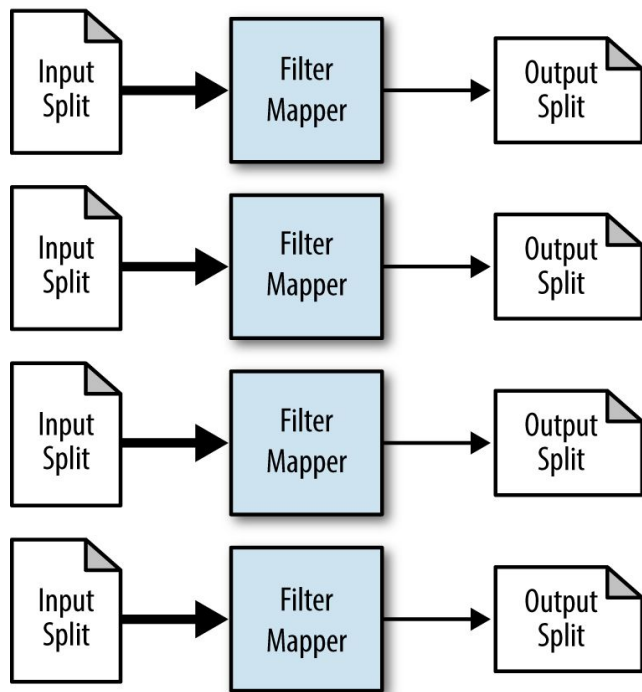
Cíl: Profiltrovat data tak, abychom dostali ta, která jsou pro nás zajímavá a naopak se zahodila ta, která jsou nezajímavá. Jedné se vlastně o hromadnou aplikaci funkce f , která na záznam vrací `true` nebo `false` podle toho, zda je pro nás záznam zajímavý

Motivace: Prohledáváme záznamy o plánovaných eventech a zajímají nás pouze takové, které mají něco společného s `BigData`. Slovo `BigData` je buď přímo v textu a nebo je event přímo otagován jako `"BigData"`. Po vybrání pouze podmnožiny dat, která nás zajímá pokračujeme s analýzou pouze v této, výrazně menší množině.

Snažíme se o něco podobného jako je SQL:

```
SELECT * FROM table WHERE value <3;
```

Filtrační patterny – filtr



Příklady:

- Distribuovaný grep
- Tracking vlákná událostí – procházíme všechny logy na našem serveru a vybíráme záznamy, kde je uživatel Bobby Green
- Předčištění dat – procházíme záznamy a na výstup postupujeme pouze ty, které mají kompletní data
- Vybíráme náhodné vzorky dat z distribuovaných úložišť, abychom mohli provést prvotní analýzu – chceme méně dat, ale rovnoměrné zastoupení

Mapovací fáze: pokud nás záznam zajímá, tak ho posíláme dále na výstup, jinak ho zahazujeme

Kombinační a redukční fáze: není

Filtrační patterny – bloom filtr

Cíl: Profiltrovat data tak, abychom dostali ta, ve kterých jsou zástupci z předem definované množiny. Ostatní se zahazují.

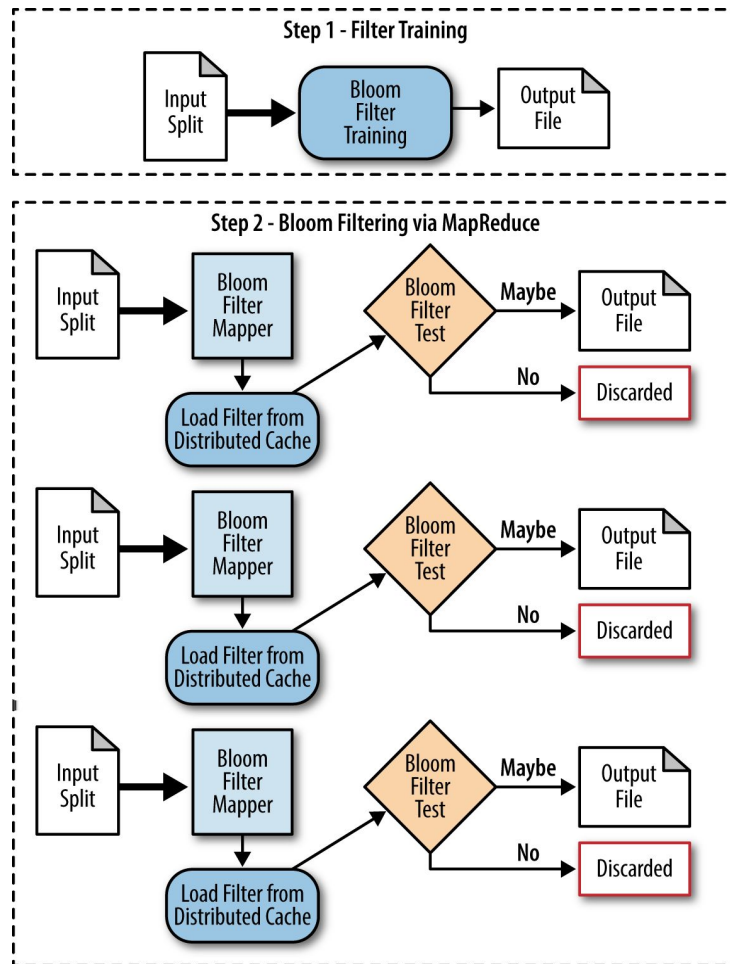
Motivace: Chceme vybrat pouze záznamy které se týkají předem definované skupiny uživatelů. Řešíme to tak, že "natrénujeme" Bloom filtr na námi definované uživatele a nahrajeme ho do distribuované cache, která je dostupná z mapper jobů.

Nebo chceme realizovat systém, který bude kontrolovat chat, aby se v něm nevyskytovala vulgární slova

Filtrační patterny – bloom filtr

Mapovací fáze: Z distribuované cache se nahraje Bloom filter - v jednoduchém případě např. seznam slova a aplikace tohoto filtru na data

Kombinační a redukční fáze: není



Filtrační patterny – top ten

Cíl: Získat relativně malou skupinu K nejlepších záznamů podle nějaké "ranking" funkce z velkého množství dat.

Motivace: Při statistické analýze dat chceme najít okrajové hodnoty (outliers) protože ty se nějakým způsobem odlišují od standardu a jsou pro nás důležité. Jestliže lze definovat ranking funkci, která je schopná porovnat dva záznamy, tak lze takto nalézt uspořádání, které funguje i přes veškeré záznamy. Tento pattern se nesnaží setřídít všechny záznamy přes všechna data, což je výpočetně extrémně náročné, ale vybrat menší množství dobře hodnocených záznamů.

Snažíme se o něco podobného jako je SQL:

```
SELECT * FROM table ORDER BY col4 DESC LIMIT 10;
```

Filtrační patterny – top ten

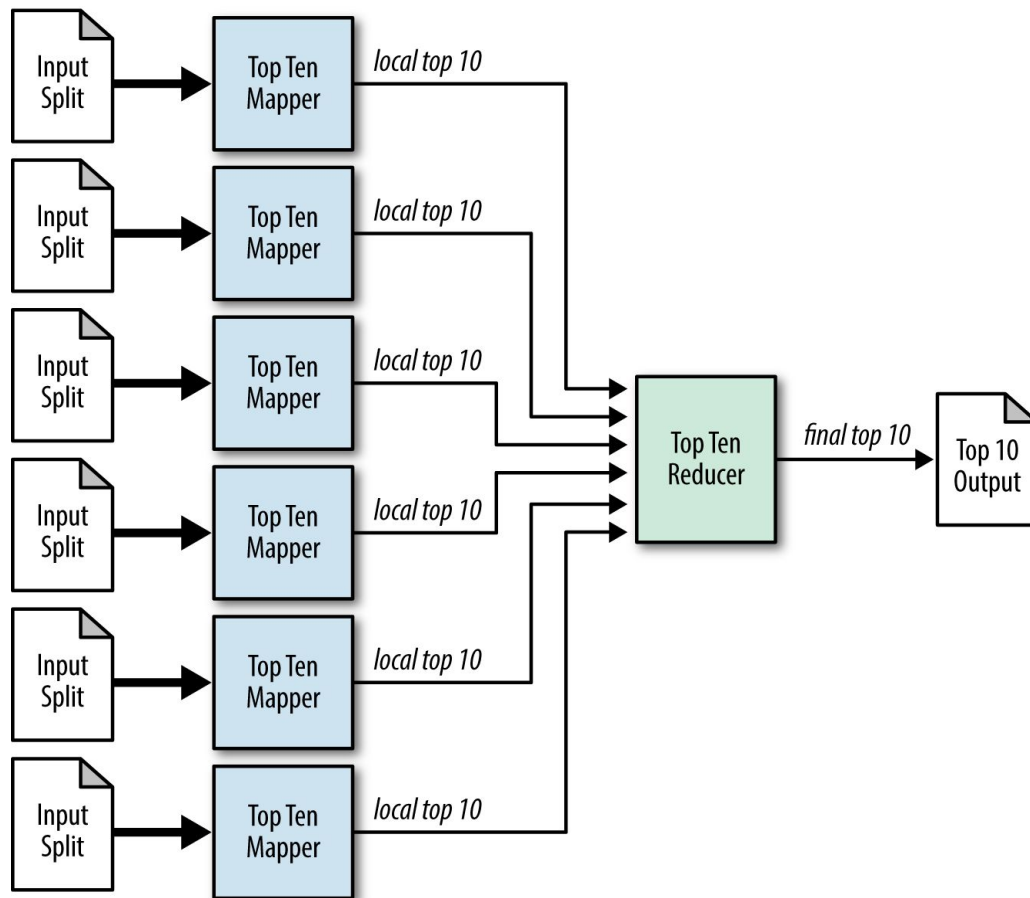
Mapovací fáze: vybírá top ten lokální záznamy

Kombinační fáze: není

Redukční fáze: Reducer setřídí záznamy a vybere finální top ten skupinu

Příklady:

- Analýza okrajových hodnot (outliers)
- Výběr zajímavých dat podle value scoring funkce
- Předčištění dat – procházíme záznamy a na výstup postupujeme pouze ty, které mají kompletní data
- Top ten produkty



Join patterny – struktura do hierarchie

Cíl: Propojení záznamů mezi sebou podle nějakého klíče

Motivace: Chceme zmigrovat data z relační databáze do dokumentově orientovaného úložiště. Máme webové stránky s blogy na různé tématické okruhy. V rámci těchto blogů se zadávají příspěvky a k nim vznikají revize. Data jsou uložena v relační databázi.

- ⇒ V případě příchodu na stránku se její obsah poskládá z různých databázových tabulek, což je náročné
- ⇒ V rámci analýzy dat chceme korelovat délku příspěvku s délkou komentářů (komplikovaná join operace a pak extrakce dat) – jestliže bychom měli data zgrupovaná podle příspěvků tak, že jsou u nich komentáře a revize, tak je analýza jednodušší.

Posts

Post

Comment

Comment

Post

Comment

Comment

Comment

Organizační patterny – struktura do hierarchie

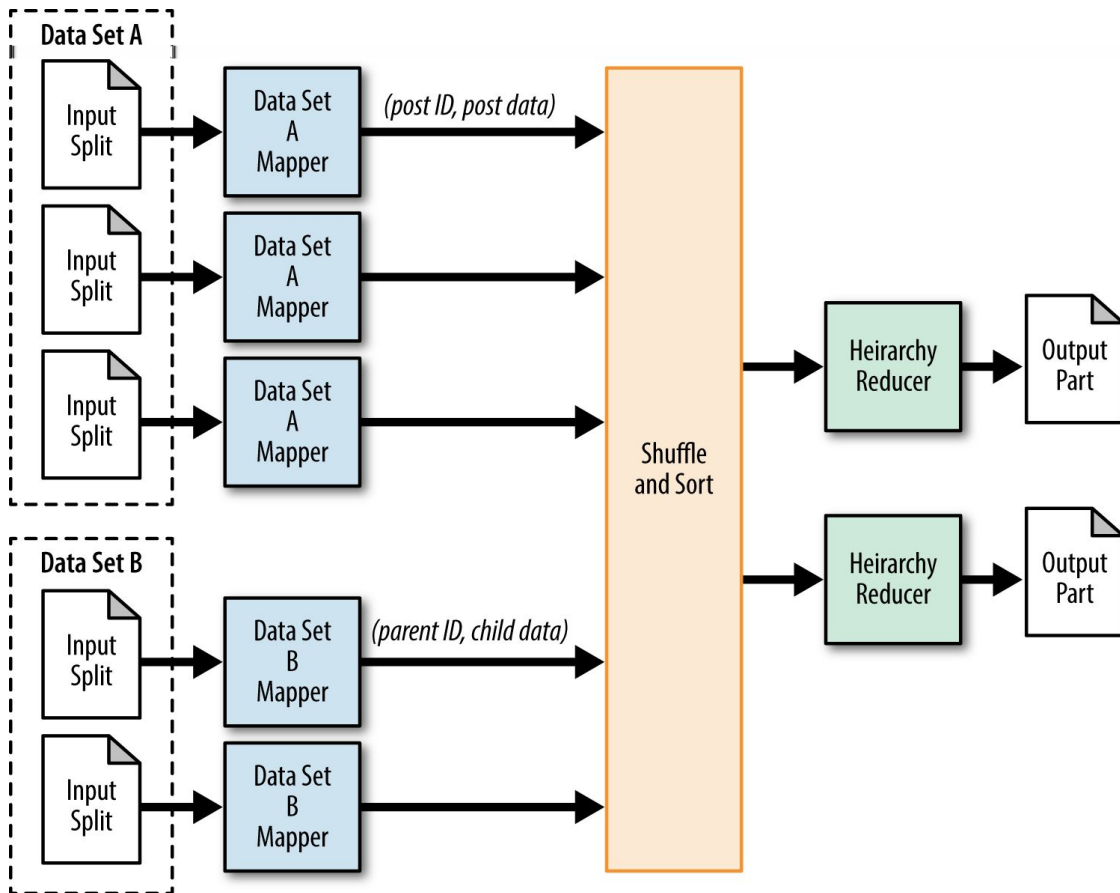
Mapovací fáze: výstupní klíč odpovídá tomu jak chceme identifikovat kořen naší hierarchické struktury (např. postId, parentId=postId). Typ dat lze identifikovat např. tím, že vlastním datům dáme prefix (P pro Post, C pro Comment)

Kombinační fáze: nemá zde příliš význam

Redukční fáze: Reducer přebírá kolekci záznamů, které mají stejný klíč (tedy kolekci posts a comments se stejným postId). Při iteraci pak přebalí data do cílového formátu - např. obalí xml tagy.

Příklady:

- Transformace dat pro MongoDB (dokumentově orientovaná databáze)
- Propojení dat přes klíč



Organizační patterny – struktura do hierarchie

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = new Job(conf, "PostCommentHierarchy");
    job.setJarByClass(PostCommentBuildingDriver.class);
    MultipleInputs.addInputPath(job, new Path(args[0]), TextInputFormat.class, PostMapper.class);
    MultipleInputs.addInputPath(job, new Path(args[1]), TextInputFormat.class, CommentMapper.class);
    job.setReducerClass(UserJoinReducer.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    TextOutputFormat.setOutputPath(job, new Path(args[2]));
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(Text.class);
    System.exit(job.waitForCompletion(true) ? 0 : 2);
}
```

Organizační patterny – struktura do hierarchie

```
public static class PostMapper extends Mapper<Object, Text, Text, Text> {
    private Text outkey = new Text();
    private Text outvalue = new Text();
    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        Map<String, String> parsed = MRDPUtils.transformXmlToMap(value.toString());
        // The foreign join key is the post ID
        outkey.set(parsed.get("Id"));
        // Flag this record for the reducer and then output
        outvalue.set("P" + value.toString());
        context.write(outkey, outvalue);
    }
}

public static class CommentMapper extends Mapper<Object, Text, Text, Text> {
    private Text outkey = new Text();
    private Text outvalue = new Text();
    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
        Map<String, String> parsed = MRDPUtils.transformXmlToMap(value.toString());
        // The foreign join key is the post ID
        outkey.set(parsed.get("PostId"));
        // Flag this record for the reducer and then output
        outvalue.set("C" + value.toString());
        context.write(outkey, outvalue);
    }
}
```


Organizační patterny – struktura do hierarchie

```
public static class PostCommentHierarchyReducer extends Reducer<Text, Text, Text, NullWritable> {
    private ArrayList<String> comments = new ArrayList<String>();
    private DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    private String post = null;
    public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
        post = null;
        comments.clear();
        for (Text t : values) {
            // If this is the post record, store it, minus the flag
            if (t.charAt(0) == 'P') {
                post = t.toString().substring(1, t.toString().length()).trim();
            } else {
                // Else, it is a comment record. Add it to the list, minus the flag
                comments.add(t.toString().substring(1, t.toString().length()).trim());
            }
        }
        // If there are no comments, the comments list will simply be empty.
        // If post is not null, combine post with its comments.
        if (post != null) {
            // nest the comments underneath the post element
            String postWithCommentChildren = nestElements(post, comments);
            context.write(new Text(postWithCommentChildren), NullWritable.get()); // write out the XML
        }
    }
}
```

Organizační patterny – struktura do hierarchie

```
private String nestElements(String post, List<String> comments) {
    DocumentBuilder bldr = dbf.newDocumentBuilder(); // Create the new document to build the XML
    Document doc = bldr.newDocument();
    Element postEl = getXmlElementFromString(post); // Copy parent node to document
    Element toAddPostEl = doc.createElement("post");
    copyAttributesToElement(postEl.getAttributes(), toAddPostEl); // Copy the attributes to the new element
    for (String commentXml : comments) { // For each comment, copy it to the "post" node
        Element commentEl = getXmlElementFromString(commentXml);
        Element toAddCommentEl = doc.createElement("comments");
        copyAttributesToElement(commentEl.getAttributes(), toAddCommentEl);
        toAddPostEl.appendChild(toAddCommentEl); // Add the copied comment to the post element
    }
    doc.appendChild(toAddPostEl); // Add the post element to the document
    return transformDocumentToString(doc);
}

private Element getXmlElementFromString(String xml) {
    DocumentBuilder bldr = dbf.newDocumentBuilder();
    return bldr.parse(new InputSource(new StringReader(xml))).getDocumentElement();
}

private void copyAttributesToElement(NamedNodeMap attributes, Element element) {
    ...
}

private String transformDocumentToString(Document doc) {
    ...
}
}
```

Organizační patterny – struktura do hierarchie

```
private String nestElements(String post, List<String> comments) {
    DocumentBuilder bldr = dbf.newDocumentBuilder(); // Create the new document to build the XML
    Document doc = bldr.newDocument();
    Element postEl = getXmlElementFromString(post); // Copy parent node to document
    Element toAddPostEl = doc.createElement("post");
    copyAttributesToElement(postEl.getAttributes(), toAddPostEl); // Copy the attributes to the new element
    for (String commentXml : comments) { // For each comment, copy it to the "post" node
        Element commentEl = getXmlElementFromString(commentXml);
        Element toAddCommentEl = doc.createElement("comments");
        copyAttributesToElement(commentEl.getAttributes(), toAddCommentEl);
        toAddPostEl.appendChild(toAddCommentEl); // Add the copied comment to the post element
    }
    doc.appendChild(toAddPostEl); // Add the post element to the document
    return transformDocumentToString(doc);
}

private Element getXmlElementFromString(String xml) {
    DocumentBuilder bldr = dbf.newDocumentBuilder();
    return bldr.parse(new InputSource(new StringReader(xml))).getDocumentElement();
}

private void copyAttributesToElement(NamedNodeMap attributes, Element element) {
    ...
}

private String transformDocumentToString(Document doc) {
    ...
}
}
```

Join patterns – kartézský součin

Cíl: Propojení záznamů mezi sebou podle nějakého klíče

Motivace: Potřebujeme udělat párovou analýzu přes všechny záznamy. Tedy např. Potřebujeme spočítat vzájemnou korelaci pro všechny možné dvojice záznamů

