Index construction and MapReduce

Hans Philippi
(partially based on the slides from the Stanford course on IR)

March 30, 2023

MapReduce: an example of NoSQL data management

- Data model: key-value pairs
- Massive parallellism on ...
- ... large amounts of commodity hardware
- Case: index building for text collections, especially or for the Web

Index for text collections

In general for document collections:

INPUT: a term

OUTPUT: the corresponding postings list, i.e. all documents in the text collection containing this term

Specifically for the Web:

INPUT: a term

OUTPUT: the corresponding postings list, i.e. all URLs of web pages containing this term

Creating postings lists

Input : document collection <docid, text>

< 2013, "de dag die je wist dat zou komen is eindelijk hier" >
< 1980, "de do do do, de da da da" >
< 1971, "jaren komen en jaren gaan" >
< 1994, "we komen en we gaan" >

Output : a set of *postings lists* for this collection of documents

...
<"dag", [2013] >
<"de", [1980, 2013] >
<"die", [2013] >
<"do", [1980] >
<"en", [1971, 1994] >
<"gaan", [1971, 1994] >
...
<"komen", [1971, 1994, 2013] >

...

Index for text collections

INPUT: a term
OUTPUT: the corresponding postings list, i.e. all occurences of this term in the text collection containing this term.

Two steps:

- Use a tree structure (B-tree, suffix tree) that connects a term to the corresponding postings list
- Return the postings list

Postings list sorted

Query $=$ $term_1$ AND $term_2$

1. locate postings list p_1 for $term_1$
2. locate postings list p_2 for $term_2$
3. calculate the intersection of p_1 and p_2 by list merging

$term_1 \Longrightarrow$	1	3	7	**11**	37	**44**	**58**	112	...
$term_2 \Longrightarrow$	2	4	**11**	25	**44**	54	55	**58**	...

Index construction: two approaches

- Algorithms dealing with limited main memory, based on external sorting. Output of sorting phase enables index building.
- Index building based on MapReduce: generic architecture for and approach to large scale parallellism

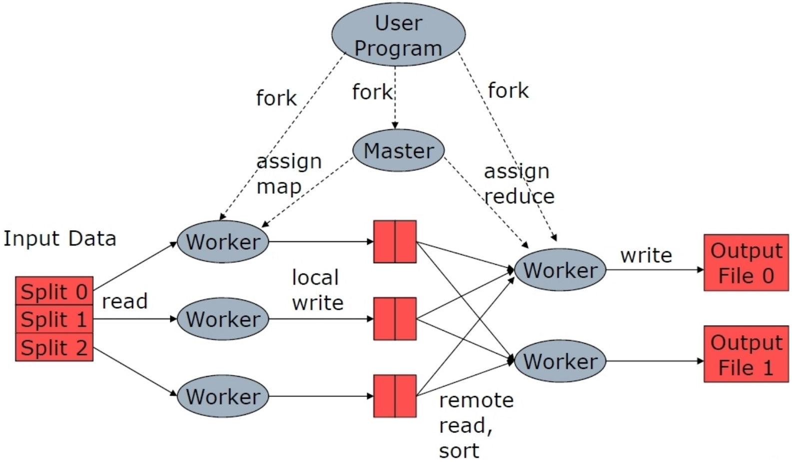
MapReduce

- Framework for massively parallel computing
- Roots in Google environment (indexing, PageRank)
- Based on commodity hardware
- Two sets of machines involved in parallel processing: *Map* workers and *Reduce* workers
- Robustness by replication of data in file system
- Generic, based on *Map* and *Reduce (Fold)* from functional programming
- Several implementations, Hadoop is the most well known

MapReduce: the Map

- Basic data structure is key-value pair $< k, v >$
- Input is split into disjoint chunks, containing collections of key value pairs
- Each Map worker works autonomous from other map workers ("shared nothing")
- Each Map worker scans it's own input chunk once
- Each Map worker does one uniform calculation on each key-value pair
- The output of each Map worker is a set of key-value pairs: zero, one or more
- The structure of the resulting key-value pairs is generally different from the input pairs
- The output results of all Map workers are collected for further processing in the Reduce phase

MapReduce computing



MapReduce: the Reduce

- The output results of all Map workers are grouped on the key values and assigned to the reduce workers
- All related key value pairs will be processed by one Reduce worker
- Each Reduce worker works autonomous from other Reduce workers (shared nothing)
- The output results of all Reduce workers together are the result of the calculation

MapReduce example

Example: *word count*

Input: a collection of documents
Output: the words in the documents with their frequency

- Map $< docid, text >$:
  for each word w in $text$
  $emit(< w, 1 >);$
- Reduce $< w, vlist >$:
  int $sum = 0;$
  for each v in $vlist$
  $sum + +;$
  $emit(< w, sum >);$

MapReduce example

Input to Map-workers:

$<$ 2013, "de dag die je wist dat zou komen is eindelijk hier" $>$
$<$ 1980, "de do do do, de da da da" $>$
$<$ 1971, "jaren komen en jaren gaan" $>$
$<$ 1994, "we komen en we gaan" $>$

Output from Map workers:

$<$"de", 1 $>$
$<$"dag", 1 $>$
$<$"die", 1 $>$
...
$<$"gaan", 1 $>$

MapReduce example

... then comes the invisible step ...

... which could be characterized as a "GROUP BY key" ...

MapReduce example

Input to Reduce-workers:

$<$"de", $[1] >$

...

$<$"komen", $[1, 1, 1] >$

...

$<$"gaan", $[1, 1] >$

...

Output:

$<$"de", $1 >$

...

$<$"komen", $3 >$

...

$<$"gaan", $2 >$

...

MapReduce example

Observations:

- The input pairs will be processed by different Map-workers
- Behind the scenes (invisible step), all emitted pairs with the same key are grouped together (after the Map phase and before the Reduce phase)
- The *grouping phase* includes concatenation of all the values corresponding to the same key
- In our example: in the grouping phase: three times $<$"komen"$, 1 >$ becomes $<$"komen"$, [1, 1, 1] >$

INTERMEZZO MapReduce example

Do you have any suggestions for optimization of the MapReduce program from the example on slide 12?

MapReduce computing: early combining

- Word count could be optimized by doing some aggregation in the Map phase
- Instead of k repetitions of $emit(< w, 1 >)$; do $emit(< w, k >)$;
- Adapt the Reduce program (how?)
- In general, this idea is applicable if the reduce function is commutative and associative (e.g. sum, max)
- Early combining often requires a setup of local datastructures and a final emit
- Our convention: for writing pseudo code, use functions $Init\_Map()$ and $Finalize\_Map()$

Word count speed up

- Init_Map():

  Create a dictionary D (word, freq);
- Map $< docid, text >$:

  for each word w in $text$

  add w to D;
- Finalize_Map():

  for each entry (word, freq) in D

  $emit(< word, freq >)$;
- Reduce $< w, vlist >$:

  int $sum = 0$;

  for each v in $vlist$

  $sum+ = v$;

  $emit(< w, sum >)$;

Let's do it

Schrijf pseudocode voor Map en Reduce voor een collectie tupels van de vorm `<g,v>` die de volgende SQL query representeert:

```
SELECT g, SUM(v) FROM Input
WHERE v >= 100
GROUP BY g
HAVING SUM(v) >= 10000
```

Let's do it again

Schrijf pseudocode voor Map en Reduce voor een collectie tupels van de vorm `<g,v>` die de volgende SQL query representeert:

```
SELECT g, SUM(v) FROM Input
WHERE v >= 100
GROUP BY g
HAVING SUM(v) <= 10000
```

MapReduce: references

- http://infolab.stanford.edu/~ullman/mmds/ch2.pdf