Data-analysis and Retrieval
Index construction and MapReduce

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

May 4, 2022

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

Hardware characteristics

Current characteristics for commodity hardware:

	memory	disk	SSD
size	16 GB	4 - 8 TB	0,5 - 1 TB
access time	100 nsec	5 - 10 msec	0.1 msec

- Average characteristics of disk access can be enhanced by clustering
- Disk IO is block (page) based; typical block size is 8 - 256 kB

Classical approach: external sorting

Input : document collection <docid, text>

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

Output from sorting phase is basis for building index and postings lists:

<"dag", 2013 >
<"de", 2013 >
. . .
<"en", 1971 >
<"en", 1994 >
. . .
<"komen", 1971 >
<"komen", 1994 >
<"komen", 2013 >

. . .

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
- Robust
- 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 output results of all Map workers are collected for further processing in the Reduce phase

MapReduce: the Reduce

- The output results of all Map workers are grouped on the key values
- After regrouping, the resulting key-value sets are distributed over 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: a formal view

- Two step approach
- *Map phase:* define a function *Map* taking $< k, v >$ as argument, finally emitting zero, one or more key-value pairs $[< k_1, v_1 >, < k_2, v_2 >, \dots, < k_m, v_m >]$
- *Reduce phase:* define a function *Reduce* taking $< k', [v'_1, v'_2, \dots, v'_n] >$ as argument, finally emitting zero, one or more key-value pairs $[< k'_1, v''_1 >, < k'_2, v''_2 >, \dots, < k'_{n'}, v''_{n'} >]$

Take notice of the accents and subscripts: they are essential

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" $>$
$<$ 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:

$<"de", 1>$

...

$<"komen", 3>$

...

$<"gaan", 2>$

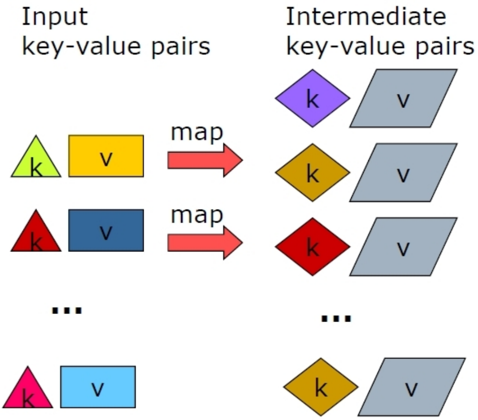
...

MapReduce example

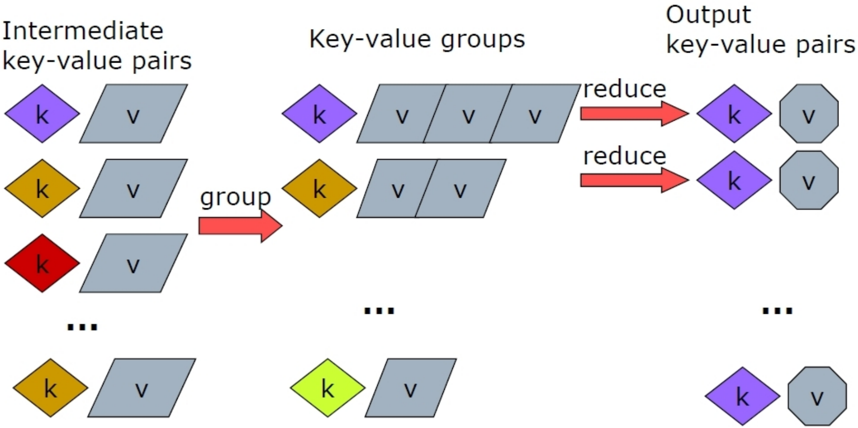
Observations:

- 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 $<"\text{komen}", 1>$ becomes $<"\text{komen}", [1, 1, 1]>$

MapReduce computing



MapReduce computing



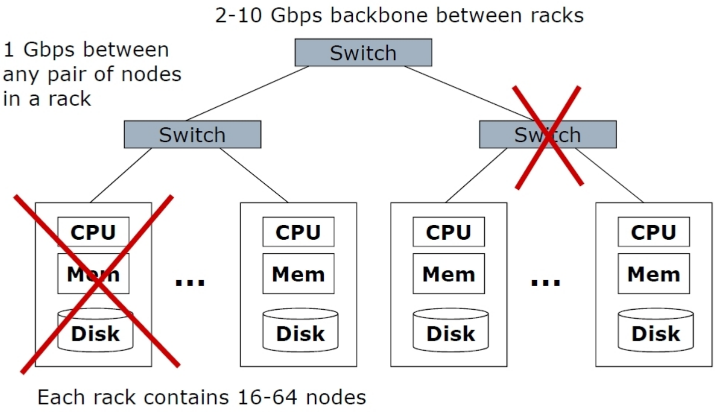
INTERMEZZO MapReduce example

Do you have any suggestions for optimization of the MapReduce program from the example on slide 9?

MapReduce architecture (original paper)

- We are dealing with terabyte scale processing problems
- Standard shared-nothing architecture
  - cluster of commodity Linux nodes
  - Gigabit ethernet interconnection
  - cheaper than supercomputer
- Masking hardware failures
- Input and final output on distributed file system

MapReduce architecture (original paper)

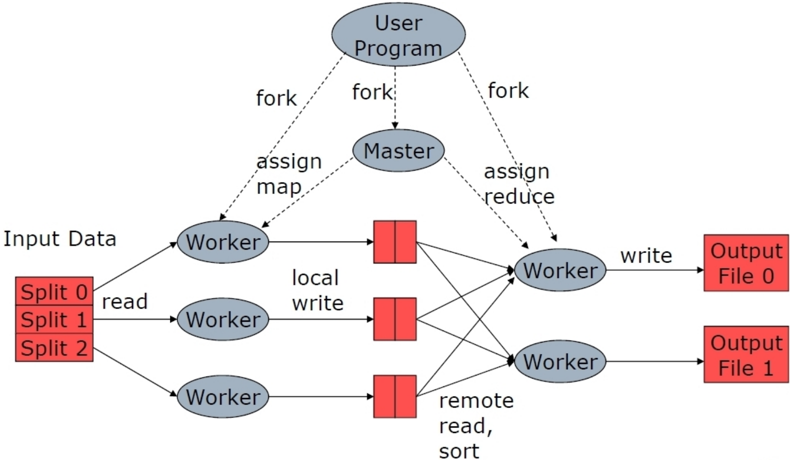


MapReduce architecture: DFS

Distributed file system

- gigabyte to terabyte scale
- data warehouse behaviour
    - read intensive
    - rarely updated
    - possibly appends
- file is split into 16-64 MB contiguous chunks
- 2-3 times replicated in different racks

MapReduce computing



MapReduce computing: coordination

Master process

- determines number of Map tasks (M) and number of Reduce tasks (R)
  - M and R are chosen much larger than the number of nodes
- monitors status workers: *idle, busy, down*
- monitors status tasks: *idle, in-progress, completed*

After finishing, a Map task delivers R intermediate result files on the local disks of the Map workers and sends the sizes and locations to the Master

- Master forwards this info to the reduce workers

MapReduce computing: failure handling

Master pings workers periodically to detect failures

- If Map worker fails, completed tasks or tasks in-progress are set to *idle*; tasks are rescheduled to other workers
- If Reduce worker fails, its in-progress tasks are set to idle
- If Master fails, job is aborted and client is notified

MapReduce computing: partitioning

- Each Map task deals with a contiguous segment of input file
- The intermediate records with the same key should end up at the same Reduce worker
- System uses a default partition function: hash(key) mod R
- It is possible to override the default partition function

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 $Finalize\_Map()$

MapReduce: caveats

- In our examples, the input of Map sometimes ignores the $< key, value >$ structure, because the keys are irrelevant. For instance, an input of value v is given, where, strictly spoken, it should be $< k, v >$.
- Often, the output of Reduce ignores the $< k, v >$ structure.
- However, for the communication between Map and Reduce, the $< k, v >$ structure is essential! The keys and values cannot be complex datastructures.
- Be aware dat Map and Reduce workers have *no direct access* to each other's local data!
- Input: Map always works on one $< k, v >$ tuple. Reduce always works on one $< k, [v_1, v_2, ..., v_n] >$ tuple.
- The only output method is *emit*.

MapReduce computing: exercises

See web site and/or handouts

MapReduce: references

- http://infolab.stanford.edu/~ullman/mmds/ch2.pdf up to and including 2.3
- https://sites.google.com/site/mriap2008/lectures