Ciência de Dados em Larga Escala

Inês Dutra and Zafeiris Kokkinogenis

DCC-FCUP
room 1.31
ines@dcc.fc.up.pt
zafeiris.kokkinogenis@gmail.com

23/24

MapReduce model

(Based on MapReduce: Simplified Data Processing on Large Clusters)

- Motivation: need for many computations over large/huge sets of data
- Computations can be done in parallel
- Complex to manage: race conditions, debugging, data distribution, fault-tolerance, load balancing etc

MapReduce model

Abstraction that allows to express simple computations but hiding the messy details of parallelization, fault-tolerance, data distribution and load balancing

programming model + library

MapReduce example: word count

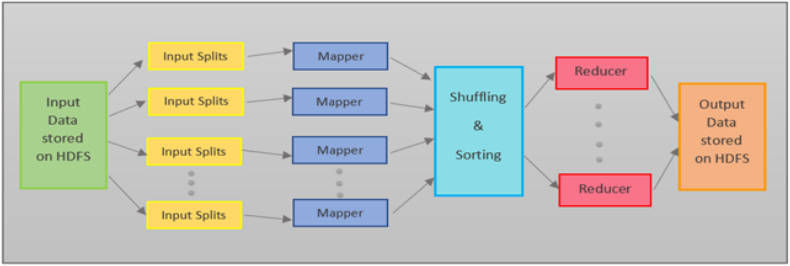
```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
     EmitIntermediate(w, "1");

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(key, AsString(result));
```

MapReduce example: word count

In general:

- Map task: a single pair → a list of intermediate pairs
  map(input-key,input-value) → list(out-key, intermediate-value)
  $\langle k_i, v_i \rangle \rightarrow \{k_{int}, v_{int}\}$
- Reduce task: all intermediate pairs with the same k_{int} → a list of values
  reduce(out-key, list(intermediate-value)) → list(out-values)
  $\langle k_{int}, \{v_{int}\} \rangle \rightarrow \langle k_o, v_o \rangle$

MapReduce: how does it work?



HDFS: Hadoop Distributed File System. It can also use GFS, the Google File System

MapReduce

- User specifies:
  - M: number of map tasks
  - R: number of reduce tasks
- Map:
  - MapReduce lib splits the input file into M pieces
  - Typically 16-64 MB per piece
  - Map tasks are distributed across the machines
- Reduce:
  - Partitioning the intermediate key space into R pieces
  - hash(intermediate key) mod R
- Typical setting:
  - 2,000 machines
  - M = 200,000
  - R = 5,000

MapReduce: fault-tolerance

- Worker failures:
  - identified by sending heartbeat messages by the master. If no response within a certain amount of time, then the worker is dead
  - in-progress and completed map tasks are rescheduled (map output is stored locally)
  - in-progress reduce tasks are rescheduled (reduce output is stored in GFS)
- Master failure:
  - Rare
  - Can be recovered from checkpoints?
  - Aborts the MapReduce computation and starts again

Disk locality

- GFS stores typically three copies of the data block in different machines
- Map tasks are scheduled close to data
  - on nodes that have input data (local disk)
  - if not, on nodes that are nearer to input data (e.g., same switch)

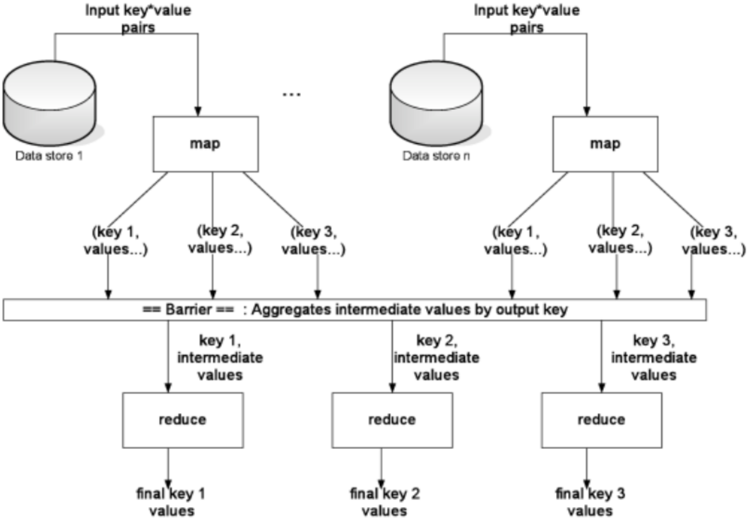
Task granularity

- Number of map tasks > number of worker nodes
  - better load balancing
  - better recovery
- but...increases master load
  - more scheduling
  - more states to be saved
- M could be chosen according to file system block size
- R is usually specified by the user (each reduce task produces one output file)

Stragglers

- Slow workers (**stragglers**) delay overall computation
- Very close to the end of the MapReduce operation, master schedules backup execution (redundancy) of the in-progress tasks
- A task is marked as complete whenever either the primary or the backup execution completes
- Google reports average improvement in job response times by 44%!
- Strategy may not work well if cluster is heterogeneous

MapReduce in a little more detail



Barrier may become a problem in the context of redundant (backup) tasks and heterogeneous clusters. Scheduler assumptions are broken.

Scheduler's Assumptions

- Nodes can perform work at roughly the same rate
- Tasks progress at constant rate all the time
- There is no cost to starting a speculative task
- A task's progress is roughly equal to the fraction of its total work
- Tasks tend to finish in waves, so a task with a low progress score is likely a slow task
- Different tasks of the same category (maps or reduces) take roughly the same amount of work

Scheduling in MapReduce

- When a node has an empty slot, Hadoop chooses one from the three categories in the following priority:
  1. A failed task
  2. Unscheduled tasks. For maps, tasks with local data to the node are chosen first.
  3. Speculative task (backup execution)

Deciding on speculative tasks

- Which task to execute speculatively?
- Hadoop monitors tasks progress using a *progress score*: a number in the interval [0,1] that measures each task's progress compared with the average progress
- For mappers: the score is the fraction of input data read
- For reducers: the execution is divided into three equal phases, $\frac{1}{3}$ of the score each:
  - Copy phase: percentage of maps that output has been copied from
  - Sort phase: percentage of data merged
  - Reduce phase: percentage of data passed through the reduce function

    Example1: $1/2*1/3$, progress score of a task halfway through the copy phase

    Example2: $1/3 + 1/3 + 1/2*1/3 = 5/6$, progress score of a task halfway through the reduce phase

Deciding on speculative tasks

- Based on average progress of each category and threshold: When a task's progress is less than the average for its category minus 0.2, and the task has run at least one minute, it is marked as a straggler:
  threshold = avgProgress – 0.2
- All tasks with progress score < threshold are stragglers
- Ties are broken by data locality
- This approach works reasonably well in homogeneous clusters

Improvement

- *progress rate* instead of *progress score values*
- backup tasks with low progress rate that are “far enough” below the mean

$$progress\ rate = \frac{progress\ score}{execution\ time}$$

Tutorial

MapReduce tutorial Hadoop 3.3.6