TU München, Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Prof. Dr. Thomas Neumann

Exercises for *Foundations in Data Engineering*, WiSe 22/23

Alexander Beischl, Maximilian Reif (i3fde@in.tum.de)

http://db.in.tum.de/teaching/ws2223/foundationsde

Sheet Nr. 10

Exercise 1

1. Implement iterative PageRank computation in pseudo code on a directed graph. Instead of an informed termination criterion, just do 20 iterations.

$$PR_0(n) = \frac{1}{graph.nrNodes} \quad \text{(init)}$$

$$PR_{i+1}(n) = \frac{1-d}{graph.nrNodes} + d \sum_{o \in in(n)} \frac{PR_i(o)}{|out(o)|} \quad \text{(step)}$$

Perform one initialization step. Then in one iteration, update the PageRank PR for all nodes in the graph. Repeat 20 times. To work on the graph, these fragements may be helpful:

```
# Retreives number of nodes in the graph
nrNodes = graph.nrNodes()
# iterates over all nodes in graph
for(n : graph.nodes()) /* do something with n */
for(neigh : graph.out(n)) # iterates over the neighbors
for(neigh : graph.in(n)) # iterates over the
                         # reverse neighbors (incoming edges)
# Creates a property on the graph.
# A property stores information for every node.
pr = graph.makeProperty()
pr[n] = 5 # sets property pr for node n to 5
```

Solution: Idea: Before letting your students work on this, draw a graph on the whiteboard and demonstrate how values are passed from one node to its neighbors.

```
def pageRank(graph, d):
   nrNodes = graph.nrNodes()
   pr = graph.makeProperty()
   pr_next = graph.makeProperty()
   # write initial values
   for (n : graph.nodes())
     pr[n] = 1/nrNodes
     pr_next[n] = (1-d)/nrNodes


   for(iteration : range(1,20)):
      # for all nodes in graph
      for(n : graph.nodes())
         # compute outgoing pr fraction
         frac = pr[n] / graph.nrNeighbors(n)
         # add fraction to all neighbors
         for(neigh : graph.out(n))
            pr_next[neigh] += d * frac
      # swap next and current
      swap(pr_next, pr)
      # reset next

      for (n : graph.nodes())
         pr_next[n] = (1-d)/nrNodes
   return pr
```

This solution looks at each node in the graph, computes the fraction of PR which it has to pass on to its neighbors, and writes the fraction to each neighbor. It is of course also possible to do this the other way around: Look at one node and collect the fractions from all reverse neighbors. The latter performs better when executed on a single machine, because fewer stores to memory are issued. However, in our case it is more instructive to use the provided variant, as it fits better into the message passing question.

2. Assume this computation is performed in main memory. Which parts of the implementation exhibit good behavior for modern computers, which parts are problematic and why? Do caches help to alleviate this?

   Solution: Good behavior: Sequential reads on graph and neighbor information. Bad behavior: Random access on pr_next[neigh]. Random access takes a lot more time than sequential access. Caches: In general, they are much to small even to hold a medium sized graph. However, they are good at caching nodes with many incoming edges. These also often visited nodes, so caches can have a large impact, depending on the structure of the graph.

3. What happens when the memory requirements of your implementation exceed the available main memory?

   Solution: The system starts allocating more virtual memory than available physical memory. Rarely used pages will be swapped out. That means, whenever a node on a swapped out page is read or updated, the page needs to be fetched from disk and is

placed into main memory. Thereby displacing another page.

4. Can you think of an optimization that helps when the working set is only slightly larger than main memory?

   Solution: Sort the nodes in the graph by in-degree. That way, often accessed nodes will be on the same pages as other often accessed nodes. They have a smaller change of being evicted.

5. Create a simple adaption of your PageRank implementation to distribute the code on multiple machines. Assume that you have at least these two functions for networking available: `MachineId getMachineIdForGraphNode(Node n)` and `send(MachineId m, Message m)`.

   Solution: Simple solution: Replace pr_next[neigh] += d * frac with m = getMachineIdForGraphNode(neigh) send(m, /* Increment neigh by frac */) /* and also put some code in to handle these messages on the receiving end */

6. At which point may this implementation make inefficient use of available resources?

   Solution: Problematic:

   - Sending messages right away will send multiple messages to each machine. Lots of overhead, network congestion etc.
   - What happens when all messages can't be handled in main memory?

7. Implement the same computation with MapReduce.
   Solution:

```
map(node, pr_node):
   frac = pr_node / graph.nrNeighbors(node)
   for(neigh : graph.out(node))
      emit(neigh, frac)

reduce(node, fracs):
   pr = (1-d)/graph.nrNodes() +  d * sum(fracs)
   emit(node, pr)

main():
   # init
   initVal = 1/graph.nrNodes()
   for(node : graph.nodes()):
      emit(node, initVal)
   # run interations
   for(iteration : range(1,20)):
      mapAll(map)
      reduceAll(reduce)
```

8. How is message passing between machines handled now? How is random access addressed? How is the out-of-memory case handled?

   Solution: Message passing is now handled by the runtime system in the shuffle phase. In this phase, all message routing is handled according to the map and reduce keys. If the runtime system has an efficient implementation for this, everything is taken care of. Thus this concern is taken away from user code. The same goes for the out-of-memory case. Here, the runtime system handles writing to disk.