

Update (10/1): Q9 - important: extra explanation about MR format requirements on exam
Update (9/27): Q4 - added alternate solution for base-2 storage

Homework 1 Solutions: CS425 FA26

Note to Students: Recommended solutions for HW1 (FA26) are below. For many questions, alternate solutions are possible, and reasonable correct solutions will be accepted during grading. So please refrain from asking questions about solutions until you receive your HW grades back.

1.
 - a. Once infected, each subtree (left or right) takes $O(\log(N/2-1) = O(\log(N)))$ time to disseminate the gossip within that subtree's group. After most of the sender's subtree has been infected, the expected time to infect the root of the tree is $O(1)$ (it may actually be faster than that, but nothing is smaller than $O(1)$ anyway). Then it takes the sender $O(1)$ rounds to infect the other subtree. This is followed by another $O(\log(N/2-1) = O(\log(N)))$ to disseminate the gossip within that subtree's group. So the total time is $O(\log(N)) + O(1) + O(\log(N)) = O(\log(N))$.
 - b. For the root, each of the N non-root nodes chooses the root with probability $2/N$, so the root receives an expected $N \cdot (2/N) = 2$ messages per round. For any non-root node, there are $N/2-1$ nodes in the same subnet, and the expected number of messages it receives from them is $1-2/N$ (Each node chooses its own subnet with probability $1-2/N$, and then chooses uniformly among the $N/2-1$ nodes in that subnet. Therefore, the probability that any one of them sends a message to this particular node is $(1-2/N) / (N/2-1) \cdot (N/2-1) = 1-2/N$). It also receives a message from the root with probability $1/N$, thus, a non-root node has load $1-1/N$, so the maximum is at the root: $2=O(1)$.
2.
 - a. What is the key difference between AWS Lambda and AWS EC2?
 - i. **Ans:** Key Difference: In EC2, users manage the VMs that their program runs on. In Lambda, users only specify the program to run and resources it needs; the cloud provider manages running it on the VMs.
 - ii. (Other: EC2 is priced by the number and size of VMs used. Lambda is priced by number of function calls and resource per call. Lambda functions are generally stateless; EC2 can be used to implement stateful applications)
 - b. What are the two key differences between AWS Lambda and AWS spot instances (think: pricing and how long instances last)?
 - i. **Ans:**
 - ii. Pricing: Lambda is priced by number of function calls and resource per call. Spot instances are priced by number and type/size of spot instance for the amount of time they are running (prices keep varying)

- iii. Lifetime: Lambda lasts for as long as the function/program is running. Spot instance runs until it is preempted by the cloud provider or the user shuts it down.
 - iv. (Other: Key difference from (a) applies here as well since, spot instances are just EC2 instances)
- c. What is the key difference between a “regular” AWS instance (EC2) and a “burstable” AWS instance?
 - i. **Ans:** With regular instances, the amount of CPU cycles available to programs running on that instance is limited to the size of the instance chosen. With burstable instances, the programs can momentarily (for short durations) use more CPU than their baseline allocations using accumulated CPU credits, i.e “bursting”
- d. What is the difference between a GPU and a “TPU” (among cloud offerings)?
 - i. **Ans:** GPU has a general purpose architecture that can accelerate any highly parallelizable program. TPU was specifically designed for AI/ML applications, specifically for fast matrix multiplication (using systolic arrays) . All major cloud providers have GPU offerings. Only Google has TPU offerings
- e. What is the difference between AWS spot instances and Google Cloud preemptible instances?
 - i. **Ans:**
 - ii. AWS spot instances prices vary dynamically (multiple times a day), while GCP preemptible instance prices are more static (changes up to once every 30 days). Or, for newer GCP spot, pricing can change up to once every day.
 - iii. Source: <https://cloud.google.com/compute/docs/instances/spot>
 - iv. Note: GCP preemptible instances and spot instances have the same pricing - preemptible instances were an earlier version of spot, which could only remain running up to 24 hours. Spot instances do not have that restriction.
- f. Give one example application (class) where you would prefer AWS EC2, and one where you would prefer AWS Lambda. Justify your choices briefly
 - i. **Ans:** Typical Applications:
 - ii. Lambda: Event processing tasks; eg: User authentication, File/photo compression, Input parsing and writing to DB, etc
 - iii. EC2: Any server like application that must be “always on” (think any user facing app - uber, doordash, etc)
 1. *(That said, large applications are often a mix of EC2 and Lambda.*
 2. *General rule of thumb:*
 3. *Lambda: Short running jobs with low resource requirements. Users does not want to manage VMs etc (usually developers with no devops support)*
 4. *EC2: Dynamic mix of jobs (length of jobs, resource requirements etc). User is well versed with DevOps)*

3.

Docker, Container, Virtual Machine (VM), Kubernetes (K8s), K8s Pod, K8s cluster.

a. Define each of these 6 terms concisely (1 sentence per term).

Ans:

1. Containers: Containers are OS (user-space) virtualization that allow running programs in isolated environments while sharing the same host OS, each with its own files, dependencies and resource allocation.
2. Docker: Docker is a platform to create, modify and deploy containers.
3. VM: VM is a technology that virtualizes a physical machine to run multiple OS kernels offering full isolation between the programs running on the OSes
4. K8s: K8s is an orchestration system used to manage containers on a cluster of physical machines or VMs
5. K8s Pod: Pod is a container or a bundle of containers that is treated as a single unit by K8s for purposes like scheduling, creation/deletion, configuration updates, network namespace allocation etc.
6. K8s cluster: K8s Cluster refers to a collection of physical machines or VMs managed by K8s to run pods on them.

b. Among the four possibilities of docker, container, VM, and K8s, give one scenario where one would use each of them over the three possibilities.

Think of applications! (Keep your answer to this part to < 100 words).

Ans

1. Docker: Deploy, manage, and test a small application on a single machine with a consistent development environment, e.g., a web application or games.
2. Container: Container runtime is preferred when you want to get lower overhead, access to lower-level container internals, or more seamless interaction with other tools, e.g., a basic web server or Redis cache, for improved resource efficiency and better interaction with Kubernetes.
3. VMs: need to run applications on different platforms, e.g., testing applications in different OS platforms.
4. K8s: run large-scale distributed systems, e.g., Hadoop, Apache Spark.

c. Give two key differences between a K8s node, K8s pod, and K8s cluster.

Ans:

Difference in functions: K8s node is a physical node or a VM running an OS and can run any program. K8s pod is a container (or a bundle of containers) that is built to run one particular program or task. K8s cluster is a collection of K8s nodes (nodes need not be identical)

Difference in ownership: K8s pod is typically managed by the application developer who writes the program. K8s nodes and what they run are typically automatically managed by

an orchestrator like Kubernetes. At the highest level K8s cluster is managed by the infrastructure engineer (devops) who decides the number and types of nodes that the cluster consists of.

4.

- Notes:
 - 20 nodes, 5 containers each -> 100 tasks can be run at once
 - network transfer needed for non-colocated data
 - each reducer gets an equal share of data from each mapper
 - assuming decimal units for storage and network
 - Alternate: base-2 for storage (-ibibytes), base-10 for network
 - assuming fair network
 - barrier between map phase & shuffle+reduce
- Map:
 - 300 map tasks -> 3 rounds
 - per round:
 - map task: 20 seconds
 - total: $3 * 20 \text{ seconds} = 60 \text{ seconds}$
- Shuffle & Reduce:
 - 200 reduce tasks -> 2 rounds
 - 5 containers per node, 3 full map rounds -> each node ran 15 map tasks
 - recall from class: map task output stored on node, not in HDFS
 - so for each reduce task, use network to fetch output for the 300-15 map tasks that ran on different nodes
 - per round:
 - traffic for each reducer: $(1/200 * 25 \text{ MB}) * (300-15) = 35.625 \text{ MB}$
 - Base-2 storage: $(1/200 * 25 \text{ MiB}) * (300-15) = 35.625 \text{ MiB}$
 - 100 reducers run per round: $35.625 \text{ MB} * 100 = 3.5625 \text{ GB}$ of traffic for shuffle
 - Base-2 storage: $35.625 \text{ MiB} * 100 = 3562.5 \text{ MiB}$
 - shuffle: $3.5625 \text{ GB} / 2.5 \text{ Gbps} \rightarrow 11.4 \text{ seconds}$
 - Base-2 storage: $3562.5 \text{ MiB} / 2.5 \text{ Gbps} \rightarrow 11.954 \text{ seconds}$
 - reduce task: 35 seconds
 - total: $2 * (35 \text{ seconds} + 11.4 \text{ seconds}) = 92.8 \text{ seconds}$
 - Base-2 storage: $2 * (35 \text{ seconds} + 11.954 \text{ seconds}) = 93.908 \text{ seconds}$
- Overall:
 - $60 \text{ seconds} + 92.8 \text{ seconds} = 152.8 \text{ seconds}$
 - Base-2 storage: $60 \text{ seconds} + 93.908 \text{ seconds} = 153.908 \text{ seconds}$

5.

- a. A failed process is undetected only if every process monitoring it also fails. Its monitoring set contains $(M-1)+(K-1)+(R-1)$ processes. So including the failed process itself, $L=M+K+R-2$
- b. If all processes that ping a failed process also fail, its failure is undetected. Therefore, $L=M+K+R-2$
- c. In an asynchronous distributed system, a correct process may respond arbitrarily slowly, and messages may be delayed or dropped. Since the protocol must choose some timeout to detect failures, a correct process that responds more slowly than that timeout will be falsely marked as failed, no matter how long the timeout is. Therefore, neither option can guarantee accuracy. So the answer is no such S exists.

6.

Any gossip with a fixed fanout will require at least $O(\log(N))$ rounds to get to $O(N)$ (say 50% of N) members, as discussed in class. This is because the “best” dissemination involves a spanning tree with processes receiving duplicate gossips (this is true of push gossip of course, and it is also true of pull gossip on average; for the latter imagine a world where each node limits the number of pulls from it per gossip round). So asymptotically pull cannot be faster than push in getting to $O(N)$ infected. However, as discussed in class, once $O(N)$ nodes already have the gossip, the “tail” of the gossip (getting from $O(N)$ to “high probability” nearly N infected) is much faster in pull gossip ($O(\log \log N)$) than in push gossip ($O(\log N)$).

7.

- a. The player row will be removed immediately when T_{fail} fires; hence, the whole table and the messages will be smaller. But gossip with the failed member will re-insert it into the table; this will cause bandwidth waste and dead/alive oscillations. T_{fail} has not changed, so detection does not happen sooner.
- b. Failed player detection will be faster and simpler since there’s no wait for the suspicion timeout. But the false positive rate will increase because a slow player won’t be given a chance to declare its liveness. With that, the overhead of removing and adding a player will increase.
- c. Detection will become simpler and stateless because there’s no need to keep an ordered list and traverse it, while the average case stays almost the same. But in the worst case, a failed player may not be pinged for a long time, and the time boundary becomes probabilistic.

8.

First MapReduce Job

This job processes the dataset to find the outgoing and incoming links (following and followers) for each user. It filters for candidates that meet the first three conditions.

input: (a, b) where a follows b

```

map1(key, value) {
    output(a, (b, "OUT")) // Outgoing link from a (a follows b)
    output(b, (a, "IN")) // Incoming link to b (b is followed by a)
}

# key := user, values := list of (string, string) tuples
reduce1(key = user, values) {
    following := create list from values with value "OUT"
    followers := create list from values with value "IN"

    // Check condition iii: all users that X follows, also follow X back
    areFollowingFollowers = true if all users in following are in followers else false

    // Apply conditions i, ii, and iii
    if (len(followers) >= 10000000 && len(following) < 10 && areFollowingFollowers) {
        output(key, ("CANDIDATE", following)) // Potential answer
    }

    // Output all users and their followers to identify the special user U in the next job
    output(key, ("FOLLOWER_COUNT", len(followers)))
}

```

Second MapReduce Job

This job takes the output of the first job, sends it to a single reducer to find the global maximum (user u), and then filters the candidates to satisfy condition iv.

```

# key := user, value := ("CANDIDATE", following_list) OR ("FOLLOWER_COUNT",
followers_count)
map2(key = user, value) {
    // Send all records to a single reducer to evaluate the global maximum
    output(1, (key, value))
}

# key := 1, values := list of (user, (string, list))
reduce2(key = 1, values) {
    // Separate the candidates and the general follower data
    candidates_and_following := filter users and following from values with value
"CANDIDATE"
    users_and_counts := filter users and count from values with value
"FOLLOWER_COUNT"

    // Find the unique user 'u' with the highest number of followers
    user_with_max_follows = find user with maximum followers in users_and_counts

    // Evaluate condition iv for each candidate
}

```

```

for candidate in candidates_and_following:
    if !user_with_max_follows in candidate.following:
        output(candidate.user) // Output the final user ID
}

```

9. Solution:

Note: On this HW only, submissions that used two Map functions feeding one Reduce received credit. This will not be allowed in future HWs and exams. Each MapReduce job is one Map followed by one Reduce. To combine datasets, first put each one into a tagged format in its own job, then use an identity Map in the join job.

MR1 — read D1, key posts by post id

Map1(a, p, ts) -> emit(p, ("POST", a))

Reduce1: output identity // file format: p, ("POST", a)

Output referenced as MR1

MR2 — read D2, count likes per post

Map2(a, p, ts) -> emit(p, 1)

Reduce2(key=p, values=[1,1,...]) -> emit(p, ("LIKES", |values|))

Output referenced as MR2. D2 has no repeated (a,p) pairs, so counting lines counts distinct likers.

MR3 — read D3, tag the tennis roster

Map3(a) -> emit(a, "TENNIS")

Reduce3: output identity // file format: a, TENNIS

Output referenced as MR3

MR4 — join MR1 and MR2 on post id

Map4: output identity

```

Reduce4(key=p, values) {
  if no ("POST", a) entry in values: return    // like on a post not present in D1
  a = poster from the ("POST", a) entry      // exactly one, post ids are globally unique
  likes = c if ("LIKES", c) in values else 0  // a post with no likes has no MR2 entry
  emit(a, likes)
}

```

Output referenced as MR4

MR5 — join MR4 and MR3 on user id, and total the likes

Map5: output identity

```

Reduce5(key=a, values) {
  if "TENNIS" not in values: return           // a is not a US tennis player
  total_likes = sum of the numeric entries in values
  emit(a, total_likes)
}

```

Output referenced as MR5. One record per user id, so nothing downstream can duplicate.

MR6 — top K

Map6(key=a, value=total_likes) -> emit(1, (a, total_likes))

// Reduce6 has a single reduce task; no parallelism in this stage.

```

Reduce6(key=1, values=[(a, total_likes), ...]) {
  topK = get top K user ids from values by total_likes // ties broken by user id
  emit(topK)                                           // one user id per line
}

```

Notes on the design:

Every Map stage reads exactly one dataset. D1, D2 and D3 each get their own MapReduce, which is why MR1 and MR3 carry identity reducers. Their only job is to materialize the dataset so the join stages read intermediate output instead of raw input.

A tennis player who never posted reaches Reduce5 with only the "TENNIS" tag and gets a total of 0. Posts with zero likes carry 0 through MR4. Both cases keep the output at exactly K lines even when few players have any likes. Self-likes are counted. The problem never asks to exclude them. If the roster is large enough that Reduce6

becomes a bottleneck, chain one extra MapReduce that takes a local top K per bucket first, then merges those in a single reducer.

10.

(a) One BPE iteration

The corpus is stored as one line per sentence. One iteration takes 4 MapReduce jobs.

Job 1

Map1(lineID, value):

tokens = value.split() // it is okay if you skip this and assume value is a list of tokens

n = length(tokens)

for j = 0 to n-2:

emit((tokens[j], tokens[j+1]), 1)

Reduce1(pair, counts):

total = sum(counts)

emit(pair, total)

Job 2: globally most frequent adjacent token pair

Map2(pair, count):

emit("BEST", (pair, count)) // single key: everything goes to one reducer

Reduce2("BEST", list):

pair = pair with the highest count in values

if there is a tie, choose the lexicographically smaller pair

emit("merge", ("rule", pair[0], pair[1]))

Job 3: prepare the corpus

Map3(lineID, value):

emit("merge", ("sentence", lineID, value))

Reduce3("merge", values)

for each sentence in values:

```
emit("merge", sentence)
```

Job 4: rewrite the corpus

// Input consists of the outputs of Jobs 2 and 3.

Map4: Identity

Reduce4("merge", values)

```
merge_rule = tuple in values with tuple[0]=="rule"
```

```
sentences = entries in values with tuple[0] == "sentence"
```

```
for each sentence in sentences:
```

```
    lineID = sentence[1]
```

```
    tokens=sentence[2].split()
```

```
    n = length(tokens)
```

```
    out = []
```

```
    j=0
```

```
    while j < n:
```

```
        if j+1<n and tokens[j]==merge_rule[1] and tokens[j+1]==merge_rule[2]
```

```
            out.append(concatenate(merge_rule[1],merge_rule[2]))
```

```
            j = j + 2
```

```
        else:
```

```
            out.append(tokens[j])
```

```
            j=j+1
```

```
    emit(lineID, out)
```

Another approach for (a): We could manually create N partitions by having R2 emit the same merge rule N times, each with a different partition ID. M3 could assign each sentence to one of the N partition IDs using $\text{hash}(\text{lineID}) \bmod N$. This would allow Job 4 to perform the merges in parallel.

(b) To run K iterations: MapReduce doesn't inherently have a loop. The loop should be coded outside of the MapReduce framework, in the program submitting the jobs. The main program runs the four MapReduce jobs as shown above iteratively. Between iterations, the data passes through the HDFS. Iteration i reads corpus_i and writes the rewritten version to corpus_(i+1), which is read by the next iteration. The final vocabulary is the initial vocabulary plus all tokens created by the K merge rules output by MR2 across the K iterations and stored in HDFS.

(c) MapReduce is not a good framework for BPE.

1. MapReduce's execution model requires materializing outputs to HDFS to be read by the next MapReduce job. Thus, for the iterative nature of BPE, it requires repeatedly materializing to HDFS and reading from HDFS again in the next iteration, which makes 10s of thousands of iterations very inefficient.
2. Within an iteration, while counting is massively parallel, finding the max requires a single reducer. Further, the corpus rewrite is also not parallel, as it requires sending the newly

created token rule (merge rule) and all sentences to one reducer in order for it to apply the merge rule (unless one explicitly creates partitions and sends the merge rule to all partitions like in the alternate solution for (a)).