Deree
School of Graduate and Professional Education

Big Data Architectures
Winter 2025

- Final Exam – Question 5

Question 5.1

[a]

One advantage of using chunk replication is fault tolerance. When a part of the cluster/network is down and a chunk is not available, there are alternative locations where the same chunk has been saved, so it can still be retrieved. Due to the same reason, the chunks can be distributed across different nodes, allowing for parallel processing and higher efficiency of our applications due to making more efficient use of our computational resources.

A con is that it cannot work efficiently with small files (each file will be broken down to at least one 'chunk', and a chunk is 64 MBs big). This limits the scope of problems we can work on, as we need to only consider cases where there are few, large data files. Moreover, HDFS is bad for low latency access, due to the communication/network overhead incurred for the different operations to take effect across the network.

Thus, in order for an application to justify making use of chunk replication, the marginal benefit it will incur from fault-tolerance and higher efficiency through the enabling of concurrent execution need to be larger than the marginal cost which is the communication/network overhead that the app's operations will incur.

[b]

Replication means that a single block of data is saved multiple times across different locations. This is what gives fault-tolerance and speed enhancements to systems like HDFS, because if the location (i.e. the machine or node) where a replicated block is stored goes down, then it can retrieve it from somewhere else. For example, we see replication being implemented in the 'chunk replication' that HDFS uses.

Partitioning means that a data object (like a document, or a datafile) is broken down into different disjoint blocks. This is what HDFS does when it breaks down big datafiles into 'chunks' of data.

[c]

HDFS will produce 1_000_000 / 65_536 = 15.26 partitions, which will actually be manifested as 16 partitions, since we cannot have fractional partitions.

Since the replication factor is 3, it means that each partition will be replicated 3 times across the network. So the total number of chunks in the network will be equal to the number of partitions multiplied by the replication factor.

As a result, it will be equal to 16 * 3 = 48.

Question 5.2

[a]

Since there is no synchronization between them, we have no control over which thread will write its result first.

So it is possible (in a way that is out of the programmer's control) one thread manages to always print before the other.

The first thread can print 'a' 10 times in a row before the second thread gets a chance to print 'b'. Similarly, the second thread can print 'b' 10 times in a row before the first thread gets a chance to print 'a'.

Therefore, the maximum number of consecutive 'a' or 'b' that can be printed is 10.

To showcase the aforementioned behavior, we could run the following code many times to see how the results change.

```
# A simple 'threading'-module example to show this
# import threading
# N = 10
# def print_a():
#     for i in range(N):
#         print('a')

# def print_b():
#     for i in range(N):
#         print('b')

# t1 = threading.Thread(target=print_a)
# t2 = threading.Thread(target=print_b)

# # Start the threads
# t1.start()
# t2.start()
# # Wait for both threads to finish
# t1.join()
# t2.join()
```

```
# The output will be a mix of 'a' and 'b', but the
maximum consecutive 'a' or 'b' is 10.
# This is because the threads run concurrently and can
interleave their output.
```

[b]

On the other extreme, it could be the case that the threads happened to actually print each after the other each time, so we got one 'a' after each 'b', and vice versa. Thus, the minimum number of 'a' or 'b' is 1.

[c]

I consider this problem to be a reduction of the classic bounded-buffer ("producer-consumer") problem, where:

- Producers produce items into a buffer.

- Consumers consume items from the buffer.

- Mutual exclusion and full/empty signalling are required to ensure correctness.

Thus, I will implement a semaphore-based solution.

Here, the buffer is only '1' element big. I consider the thread(s) printing ‘a’ to be producers, adding to the buffer (aka turning its value from 0 to 1), and thread(s) printing ‘b’ to be consumers.

The lock ('mutex') ensures exclusive access to the shared variable ('buffer'). This prevents both threads from trying to modify 'buffer' at the same time, which could lead, for example, to consumers consuming invalid/incomplete data (aka printing 'b' when the producer has not had the time to print a).

The semaphores are used to impose strict rules of inter-thread communication and coordination. This way, we enforce the desired behavior, which is expressed in the following rules:

- ‘a’ goes before ‘b’. First we need to have a a printed for a b to be printed. Otherwise the consumer (aka printer of ‘b’) should wait.

- Moreover, we can have no ‘a’ or ‘b’ messages printed out one right after the other.

This way, these two synchronization primitives prevent 'race conditions' that could lead to 'state inconsistencies'.

```
import threading
```

Big Data Architectures
Winter 2025

```
mutex = threading.Lock()
buffer = 0  # this will always be equal to 0 or 1
empty = threading.Semaphore(1)  # Tracks EMPTY slots, aka
slots at the very beginning of execution or when there is
“b” printed out (starts full: BUFSIZE available)
# This means that when 'empty' is available for
acquisition, producers (threads printing “a”) have
remaining room in the buffer to write at.
# When it is not available / zero, producers should be
blocked from producing any further, until at least one
consumer prints “b”.
full = threading.Semaphore(0) # Tracks FILLED slots
(starts empty: 0 available)
# This means that when 'full' is available, then is space
for consumers to print “b”.
# When it is not available / zero, consumers should be
blocked from printing “b”, until at least one 'producer
prints “a”.


N = 10
def print_a():
    for i in range(N):
        empty.acquire()
        mutex.acquire()
        buffer = 1
        print('a')
        mutex.release()
        full.release()

def print_b():
    for i in range(N):
        full.acquire()
        mutex.acquire()
        print('b')
```

```
        mutex.release()
        empty.release()

t1 = threading.Thread(target=print_a)
t2 = threading.Thread(target=print_b)

# Start the threads
t1.start()
t2.start()
# Wait for both threads to finish
t1.join()
t2.join()
```

Question 5.3

An 'atomic' transaction is a transaction that is equivalent to a single unit. We want it to be completed in total to consider it, else we do not consider it at all.

A typical case of how lack of atomicity can lead to inconsistent states has to do with banking transactions. Consider a user withdrawing money from one of his accounts.

If we lack atomicity, then it could happen that the system is down and the withdrawal is registered, but the money is never actually added to their account.

So if they checked their mobile banking app, they would see an increased amount available, but actually in the bank's ledger this would not have been completed, and the actual available amount would be different - "inconsistent".

Question 5.4

The time the app takes to run sequentially is T1 = 20 mins.

The proportion of the code that cannot be parallelized is 1-0.8=0.2=20%.

The number of processors considered is p = 16.

The highest possible "theoretical" speedup would be:

T1 / Tp = 1 / (s + (1-s)/p) = 4 => the minimum "theoretical" type that it would take the app to run in parallel would be 20 / 4 = 5 mins.

However, we need to state that there have been over-simplifications made here.

The previous calculation ignores the communication overhead that occurs when we implement distributed calculations.

Moreover, it ignores workload imbalances that may exist across tasks.

Thus, this speedup discussed is not actually possible; this is why I called it “theoretical”.

Question 5.5

Even in our current day that concurrent programming tends to become the norm, traditional SQL databases do not implement multi-threading in processing single requests. This puts them at a great disadvantage (performance-wise) compared to NoSQL databases.

NoSQL databases implement data sharding (replicating data across servers) to implement concurrent execution. This gives them not only the aforementioned performance enhancement, but also allows them to scale horizontally without any single points of failure. In other words, there is enhanced fault-tolerance.