Homework 5

CS425/ECE428 Spring 2019

Due: Wednesday, May 1 at 11:55 p.m.

1. Optimistic Concurrency .... *18 points*

   Consider the interleaving of 3 transactions. t_i here is used to represent temporary variables internal to the transaction, while X and Y are objects that are read and updated by the transactions.

	T1	T2	T3
1	$t_1 = X$		
2		$t_2 = Y$	
3		$X = t_2 + 1$	
4			$t_3 = X$
5			$X = t_3 + 2$
6	$t_4 = Y$		
7	$Y = t_1 + t_4 + 3$		
8		$Y = t_2 + 4$	
9			$t_5 = Y$
10			$Y = t_3 + t_5 + 5$

(a) (2 points) Is this interleaving serially equivalent? Why or why not?

Solution: No, it is not serial equivalent. T1 reads X at time 1 and T2 writes to X at time 3. T2 reads from Y at 2 and T1 writes to Y at 7. These two conflicts result in non-serial-equivalent transaction.

(b) (2 points) Would this interleaving be possible with 2-phase locking, using either exclusive or reader/writer locks? Why or why not?

Solution: No, it is not possible. If using 2PL, T1's lock on X will prevent T2 from acquiring exclusive lock on X.

(c) (3 points) Suppose that timestamped optimistic concurrency was used. Write down for each step in the interleaving the updated read and write timestamp of the object being read or written. Assume that T1, T2, and T3 have timestamps 1, 2, and 3, respectively, and that the initial timestamps of each object are 0.

Solution:

	T1	T2	T3
1	$t_1 = X$ (X.rts=1)		
2		$t_2 = Y$ (Y.rts=2)	
3		$X = t_2 + 1$ (X.wts = 2)	
4			$t_3 = X$ (X.rts=3)
5			$X = t_3 + 2$ (X.wts=3)
6	$t_4 = Y$ (Y.rts=2)		
7	$Y = t_1 + t_4 + 3$ (Y.wts=1)		
8		$Y = t_2 + 4$ (Y.wts=2)	
9			$t_5 = Y$ (Y.rts=3)
10			$Y = t_3 + t_5 + 5$ (Y.wts=3)

Note that at step 7, transaction 1 should abort because it is trying to write to a value Y with a read timestamp greater than its own timestamp. If you wrote this in your solution and did not provide timestamps for steps 8–10 we will mark your solution as correct.

For the following parts, consider a modification of the transaction interleaving above:

	T1	T2	T3
1	$t_1 = X$		
2		$t_2 = Y$	
3		$X = t_2 + 1$	
4			$t_3 = X$
5			$X = t_3 + 2$
6	$t_4 = Y$		
7	$\boxed{X = t_1 + t_4 + 3}$		
8		$Y = t_2 + 4$	
9			$t_5 = Y$
10			$Y = t_3 + t_5 + 5$

(d) (2 points) Is this interleaving serially equivalent? Why or why not?

Solution: No, it is not serially equivalent. T1 reads X at time 1 and T3 writes to X at time 5. T2 writes to X at 3 and T1 writes to X at 7. These two conflicts result in non-serial-equivalent interleaving.

(e) (2 points) Explain what would happen in this interleaving using timestamped optimistic concurrency. Be detailed.

Solution: T1 will abort and rollback at time 7 because timestamp is 3, which is higher than timestamp 1.

(f) (4 points) What would happen in this example if multi-version optimistic concurrency was used? Write down for each operation which version of X and Y would be read or written, assuming transactions T1. T2, T3 had timestamps 1, 2, and 3, respectively.

Solution:

	Operation
1	T1: Read committed version X
2	T2: Read committed version Y
3	T2: Write to tentative version X(2)
4	T3: Read tentative version X(2)
5	T3: Write to tentative version X(3)
6	T1: Read committed version Y
7	T1: Write to tentative version X(1)
8	T2: Write to tentative version Y(2)
9	T3: Read tentative version Y(2)
10	T3: Write to tentative version Y(3)

(g) (3 points) Repeat the previous part assuming that T1, T2, T3 had timestamps 3, 2, 1, respectively.

Solution:

	Operation
1	T1: Read committed version X
2	T2: Read committed version Y
3	T2: Write to tentative version X(2)
4	T3: Read committed version X
5	T3: Write to tentative version X(1)
6	T1: Read committed version Y
7	T1: Write to tentative version X(3)
8	T2: Write to tentative version of Y(2)
9	T3: Read committed version Y
10	T3: Write to tentative version of Y(1)

2. Bloom Filter . . . . *5 points*
   Consider a Bloom filter with 16 slots that uses 2 hash functions. We will use a version of SHA256 based on your netid. To hash `x`, you will need to run:

```
$ echo -n <netid> <x> | openssl sha256
```

E.g.:

```
$ echo -n nikita 1 | openssl sha256
369ca22d9a6484a2109492a23ac355721b240132288436273fa16ce245ab04b0
```

If you prefer, you can use Python3 to calculate it:

```
$ python3
Python 3.7.0 (default, Aug 22 2018, 15:22:33)
[Clang 9.1.0 (clang-902.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import hashlib
>>> hashlib.sha256(b"nikita 1").hexdigest()
'369ca22d9a6484a2109492a23ac355721b240132288436273fa16ce245ab04b0'
```

We will use the first hex digit of the output (3 in this example) as the first hash function, and the second digit (6) as the second hash function. Be sure to use your own netid in place of `nikita` here.

(a) (3 points) Add the strings '1', '3', '5', . . ., '19' to this Bloom filter and write down which bits have been set. It's not too hard to do manually but you may want to write a quick script.

Solution: The following script is used to test on your result.

bloomfilter.py

```
import sys
import hashlib
bf = {k: 0 for k in range(int('0',16), int('f',16)+1)}
netid = sys.argv[1]
for i in range(1,20,2):
    res = hashlib.sha256((netid+" "+str(i)).encode()).hexdigest()
    bf[int(res[0], 16)] = 1
    bf[int(res[1], 16)] = 1
```

(b) (2 points) Test the resulting Bloom filter for the values '2', '4', '6', ..., '20'. Do you get any false positives, and if so, what are they?

Solution:

```
for i in range(2,21,2):
    res = hashlib.sha256((netid+" "+str(i)).encode()).hexdigest()
    if bf[int(res[0], 16)] == 1 and bf[int(res[1], 16)] == 1:
        print(str(i) + " is false positive!")
```

3. Cassandra Quorums . . . . *5 points*
Consider a Cassandra key that is replicated at 5 nodes in the local data center, L_1, L_2, L_3, L_4, L_5, three nodes in remote data center 1, R_1, R_2, R_3, and 3 nodes remote data center 2, S_1, S_2, S_3.

(a) (3 points) What is the minimum number of nodes that need to respond for a request configured with LOCAL_QUORUM, EACH_QUORUM, and QUORUM? (There are 3 subparts here)

Solution: LOCAL_QUORUM = 3
EACH_QUORUM = 3+2+2 = 7
QUORUM = 6

(b) (2 points) Consider all possible combinations of read and write policies selected out of the quorum options above. (E.g., EACH_QUORUM for read, LOCAL_QUORUM for write.) Which combinations would *not* result in consistent reads? Write down an example read and write set for each combination that would result in inconsistency.

Solution:

Read Policy	**Write Policy**	**Result**
LOCAL QUORUM	EACH QUORUM	Consistent
LOCAL QUORUM	QUORUM	Can be inconsistent: Read: L1,L2,L3; Write: R1,R2,R3,S1,S2,S3
LOCAL QUORUM	LOCAL QUORUM	Consistent
EACH QUORUM	EACH QUORUM	Consistent
EACH QUORUM	QUORUM	Consistent
EACH QUORUM	LOCAL QUORUM	Consistent
QUORUM	EACH QUORUM	Consistent
QUORUM	QUORUM	Consistent
QUORUM	LOCAL QUORUM	Can be inconsistent: Read: R1,R2,R3,S1,S2,S3; Write:L2,L3,L4

4. MapReduce.. *11 points*

Implement the following computations using a chain of one or more MapReduce operations.

(a) (2 points) Given a directed graph $G = (V, E)$, compute the indegree of each vertex. Input pairs (k, v) where k is a graph vertex and v is a list of its out-neighbors; i.e., for each $x \in v$, (k, x) is a directed edge in E. Output pairs (k, v) where k is a graph vertex and v is the number of nodes that have k as its neighbors; i.e., $v = |\{x|(x, k) \in E\}|$

Solution:

```
# output: (k,v) where (v,k) is a directed edge in E
def map(k,v):
  for node in v:
    emit( (node,k) )

# v will be a list of in-neighbors of k
def reduce(k,vs):
   emit( (k, len(vs)) )
```

(b) (3 points) Compute a list of all nodes reachable from a vertex in exactly two hops. Input is as above. Output is (k, v) where k is a graph vertex and v is a list of nodes reachable in two hops; i.e., $v = \{x|\exists y \text{ such that } (k, y) \in E \text{ and } (y, v) \in E\}$.

Solution:

```
# output: k: node, v: list of in and out neighbors
def map1(k, v):
  for node in v:
    emit( (node, ("in", k)) )
    emit( (k, ("out", node)) )

# output: (k, v) where v is reachable from k in two hops
def reduce1(k, vs):
  for dir1, node1 in vs:
    for dir2, node2 in vs:
      if dir1 == "in" and dir2 == "out":
        # node1, k is an edge and k, node2 is an edge
        emit( (node1, node2) )

# map2 is the identity map
def map2(k, v):
  emit( (k,v) )

# reduce2 aggregate the neighborhood
def reduce2( (k,vs) ):
  emit( (k,vs) )
```

(c) (3 points) Compute the size of a 4-neighborhood of each vertex; i.e., all the nodes reachable from from a vertex in at most 4 hops. Input as above; output is (k, v) where k is a graph vertex and v is the number of nodes reachable in at most 4 hops.

Solution: We first modify the previous solution to output the 2-neighborhood of a node. We only need to change `reduce1`.

```
# output: (k, v) where v is reachable from k in one OR two hops
def reduce1_new(k, vs):
  for dir1, node1 in vs:
    if dir1 == "in":
      # emit one-hop edges
      emit( (node1, k) )
    # as above
    for dir2, node2 in vs:
      if dir1 == "in" and dir2 == "out":
        # node1, k is an edge and k, node2 is an edge
        emit( (node1, node2) )
```

Essentially this produces a new graph with additional edges for nodes reachable in 2 hops. If we then run the chain again then we will get the desired result. I.e., the final chain is map1 $\rightarrow$ reduce1_new $\rightarrow$ map2 $\rightarrow$ reduce2 $\rightarrow$ map1 $\rightarrow$ reduce1_new $\rightarrow$ map2 $\rightarrow$ reduce2

(d) (3 points) Compute the product of two matrices, $M^{(1)}$ an $M^{(2)}$. Input is (k, v) where k is a triple (i, j, n) such that $M^{(n)}_{i,j} = v$. Output is (k, v) where k is a pair (i, j) such that $M'_{i,j} = v$, where $M' = M^{(1)} \times M^{(2)}$.

Solution: For simplicity assume that each matrix has dimensions $m \times m$. Note that $M'_{i,j} = \sum_{l=1}^{m} M^{(1)}_{i,l} \cdot M^{(2)}_{l,j}$. The first map/reduce chain will compute each of the products, the second one will add them together.

```
# output: k: (i,j,l) for which v needs to be involved in the product
def map1(k, v):
  i,j,n = k
  for x in range(1,m+1):
    if n == 1:
      emit( (i,x,j) )
    else:
      emit( (x,i,j) )

# output: k: (i,j) v: products that need to be summed
# to obtain M_{i,j}
def reduce1(k, vs):
  i,j,l = k
  emit( ((i,j), vs[0]+vs[1]) )

# identity map
def map2(k,v):
  emit(k,v)

def reduce2(k,vs):
  emit( (k, sum(vs)) )
```

□