Homework 6

CS425/ECE428

Due 11:59 p.m. May 6, 2020

1. Timestamped Concurrency . . . . 10 points

   Given two transactions, we can define a *run* (or an *interleaving* as an ordered sequence of the operations of the two transactions where each transaction's operations follows the order defined by the transaction. E.g., if $T_1 = (op_1, op_2, op_3)$ and $T_2 = (op^4, op^5, op^6)$, then $(op_1, op_2, op^4, op_3, op^5, op^6)$ is such a run.

   In a run, we say that a *context switch* occurs at step $s + 1$ if the operations in steps s and $s + 1$ are from different transactions. In the run above, there is a context switch at steps 3, 4, and 5. Any run of total length n will have between 1 and $n - 1$ context switches. The number of context switches can be interpreted to be a measure of concurrency of a run.

   In this question, you will consider two transactions:

   T_A: read A; read B; read C; write D; write F

   T_B: read A; read D; read G; write B; write E; write H

   (a) (1 point) Identify all the conflicts between these two transactions

   (b) (3 points) Show a run / interleaving, using two-phase locking with shared (reader/writer) locks, which maximizes the number of context switches. Show when locks are acquired, promoted, and released in your run.

   (c) (6 points) Show a run / interleaving, using timestamped concurrency, that maximizes the number of context switches and allows both transactions to commit. In your answer specify the timestamps of T_A and T_B, list how the read and write timestamps of each object are updated, and when writes are skipped according to the Thomas rule. Also note when one of the transaction gets added to the dependency list of the other.

2. Deadlocks . . . . 10 points

   Consider the following two transactions:

   T_C: read C, read S, read E, write C, write E

   T_D: read E, read C, write E, write C, write S

   (a) (2 points) Write down a partial interleaving that would result in a deadlock when using shared (reader/writer) two-phase locking. List which locks will be held (and in which mode—read or write) and which will be waited for by each transaction in your deadlock.

   (b) (3 points) Rosenkrantz et al. [1]proposed two strategies for avoiding deadlocks. In both cases, transactions maintain timestamps. When a transaction requests a lock that is currently held by another transaction, their timestamps are compared.

   In the first strategy, called *wait-die*, if the timestamp of the requesting transaction is newer than the timestamp of the transaction that holds the resource (i.e., $T(T_R) > T(T_H)$), then the requesting transaction aborts. In this way, a newer transaction never waits for a lock held by an older transaction.

[1]Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis. 1978. System level concurrency control for distributed database systems. ACM Trans. Database Syst. 3, 2 (June 1978), 178–198. DOI: `https://doi.org/10.1145/320251.320260`

In the second strategy, called *wound-wait*, if the timestamp of the requesting transaction is older than the timestamp of the transaction that holds the resource (i.e., $T(T_R) < T(T_H)$), the transaction that holds the resource is aborted.

List one interleaving of T_C and T_D that would result in an aborted transaction using *wait-die* but not *wound-wait*. State what the timestamps of the two transactions are in your example and when the abort happens.

(c) (3 points) List one interleaving of T_C and T_D that would result in an aborted transaction using *wound-wait* but not *wait-die*. State what the timestamps of the two transactions are in your example and when the abort happens.

(d) (2 points) List an interleaving that would result in an aborted transaction in *both* strategies. State what the timestamps of the two transactions are in your example (same timestamps must work for both strategies) and when the aborts happens.

3. MapReduce .... 10 points

(a) (3 points) Use a map-reduce chain to trace people who came in contact with each-other during an epidemic outbreak in a university town. The input to the map-reduce chain is in the following key-value format: (k, v), with $k = personID$, and v is a list of tuple $((x, y)$, *entry timestamp*, *exit timestamp*$)$, where (x, y) is the GPS coordinate of a location, *entry timestamp* is when the person appeared at that location, and *exit timestamp* is when the person left the location. If a person visited the same location more than once, the value list for that person would contain multiple entries for that location but with different timestamps. Output of the map-reduce chain must have the following format: (k, v), with $k = personID$, and v is a list of ids of people k "came in contact with". Assume that person A "comes in contact with" another person B if both A and B are in the same location at the same time for more than 5 consecutive minutes.

(b) (4 points) Use a map-reduce chain to compute the dot product of two vectors V_1 and V_2, each having a dimension of N. The input to the map-reduce chain is in the following key-value format: (k, v), with $k = (i, n)$, where $i \in [1, N]$ is the index of the vector V_n, and v is the corresponding value ($V_n[i]$). Your map-reduce chain must support proper partitioning and load-balancing across workers. In particular, assuming a vector dimension of 10000, and a hundred workers, ensure that a single worker is not required to handle more than ≈200 values at any stage.

(c) (3 points) Given a directed graph $G = (V, E)$, use a map-reduce chain to compute the set of vertices that are reachable in *exactly 3 hops* from each vertex. The input to the map-reduce chain is in the following key-value format: (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. The output must be key-value pairs (k, v), where k is a graph vertex and v is a list of vertices that are reachable in exactly three hops from k (the list must be empty if there are no vertices reachable in exactly three hops from k). For your assistance, the first map function for an exemplar map-reduce chain has been provided below. You may choose to use the same function, or design your own.

```
function MAP1($(k, v)$):
    for node in v do
        emit ( ( node , ( "i n" , k ) ) )
        emit ( ( k , ( "out" , node ) ) )
    end for
    if v is empty then
        emit ( (k, ("out", _)))
    end if
end function
```

4. 2PC and Paxos .... 10 points

In a Spanner and similar systems, a combination of two-phase commit and Paxos protocols are used. Both the coordinator and participants in 2PC are implemented as replica groups, using Paxos to achieve consensus in the group. Each replica group has a leader, so during 2PC, the leader of the coordinator group communicates with the leaders of the participant groups. Additionally, there are three points that consensus needs to be achieved: at each participant to prepare for a commit, at the coordinator to commit to a decision, and at each participant again to log the final commit.

(a) (4 points) Suppose that there is one coordinator and two participants. Each of these has a Paxos replica group of size 3. The latency *within* each group is T_1 and the latency *between* groups is T_2. Calculate the number of messages and the overall latency of the combined 2PC / Paxos protocol. Assume that there are no failures or lost messages in this and subsequent parts of this question.

(b) (2 points) At what point can the coordinator tell the client the transaction successfully committed? Calculate the latency until this point.

(c) (4 points) In Paxos, the proposer can be distinct from the acceptors. We can therefore modify the protocol to have the leader of the coordinator group act as the proposer for the participant Paxos groups as well. Calculate the latency of this modified protocol. When would it be superior?