ACS - Assignment 2

Rikke Tophøj Heinemann (DTZ190)
Marie Elkjær Rødsgaard (DCK495)
Rasmus Pallisgaard (DWP992)

December 7, 2023

1

1.1

It is possible to improve latency (decrease it) by concurrency in theory. If it is possible to perfectly parallelize the process, it is possible to speed up the process by $n =$ the number of sub-tasks executing in parallel, but many programs are difficult to perfectly paralellize and the concurrency comes with its own challenges.

1.2

Batching is the act of performing several of the same operations one time instead of doing them one at a time. *Dallying* opens up the opportunity for batching; it delays requests in the hope of either not performing them, if they are not needed anyway, or do batching.

1.3

When a web browser maintain a cache of recently visited websites. The fast-path is then happening when a user is returning to a websites, they visited earlier. It is a fast path because it involves reducing latency in re-visiting a website and thus repeating a frequent request. The trade-off could be increasing latency in other processes, e.g. when visiting the website in the first place.

2

2.1

So we want to find a name mapping scheme that could work with multiple machines K each have a storage system as a single address space with N entries. We note that N is significantly larger than K. And we also assume that the K is finite, not changing. For the abstractions mapping we think that some sort of hashing would make sense. Now we need to consider how we handle failure since if one machine crashes we do not want the whole thing to fail. For this we need something that can handle rehashing and remapping in a smart way. Consistent hashing could be an option. Here we think of all the addresses that

we need to hash to as a circle each machine will get a section of the circle that they will use. When a machine fails then only that portion that the machine was mapped to needs to be remapped. This would the be the level of fault tolerance that there would be. There would probably need to be a machine that handles the request From the different machines. That machine would then perform the READ or WRITE function. If that machine fails there should be some fail safe to give that responsibility to a new machine. How this should be done is not something we are sure how to.

With the consistent hashing each machine will have their own section in the circle making sure that there will be no collisions. This then means that multiple machines can make request at the same time making the mapping more efficient. The mapping would also run in constant time since that is the average for most hashing operations. We don't think there is a centralized component here.

2.2

So the READ and WRITE operations would be the individual machines making request. Hashing would be performed to locate the machine making the request. The machine responsible for handling the request would then perform the READ/WRITE. If it is a READ request then the responsible machine would find the data in the storage and send it back to the machine making the request. And for the WRITE when the machine is found then the responsible machine will write the data into the storage.

2.3

It is hard to say whether or not the abstraction presented is atomic or not for the READ and WRITE operations. Since it could be possible for one machine to make multiple request in theory but in practices it might be a bit extreme for only one machine to then handle all the request. since there will still be request from different machines. So For it to be more atomic there should maybe be more than one responsible machine.

2.4

The way we have chosen to describe our abstractions with consistent hashing would handle well the K not being finite and changing. Because each machine would have a section of the circle that they are responsible for. So the remapping of the machine being removed would be for only that section. The adding of a new key would also be possible and the resizing of the reactions of keys may be impacted here but possible.

3

3.1

We have included a drawing of the three precedence graphs, which can be seen a couple of pages down. We see the following for the schedules:

1. For schedule 1 there is a conflict between T1 and T2 caused by a `read->write` on X. There is a conflict between T2 and T3 caused by a `write->read` on Z. There is a conflict between T3 and T1 caused by a `read->write` on T3. Because this creates

a cycle in the graph, a no schedule is conflict-serializable if its precedence graph has cycles, schedule 1 is not conflict-serializable. If we could to change the $W(X)$ from transaction 2 to a $R(X)$ we would remove this conflict, resulting in the edge between T1 and T2 being removed. This could

2. For schedule 2 there is a conflict between T1 and T2 caused by a `read->write` on X, and one between T3 and T2 caused by a `write->read` on Z. Since this does not create a cycle, schedule 2 is conflict-serializable.

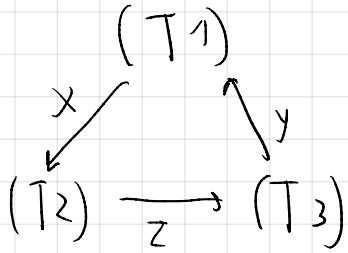
3. For schedule 3 there are two conflicts between T1 and T3 caused by a `write->read` on X and a `write->read` on Z. There are also two conflicts between T3 and T2 caused by a `write->read` on X and a `read->write` on Y. As this graph does not contain a cycle, schedule 3 is conflict-serializable.

3.2

Schedules 1 and 2 could have been generated by using strict 2PL. We have injected the shared/exclusive locks in the tables drawn below. We insert the locks in accordance with strict 2PL i.e. as needed when we encounter an operation that needs an appropriate lock. We also implicitly release all locks as the transaction is concluded (committed). These schedules are appropriate because all locks set in one transaction is released before they would create a conflict with another transaction.

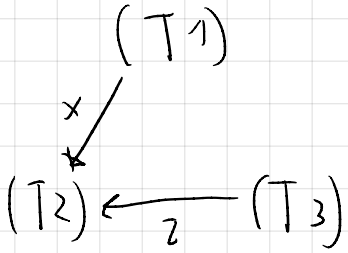
Schedule 3 is not could not have been generated by strict 2PL, and this can be seen in the very first two actions performed by transaction 1 and 3: T1 acquires an exclusive lock since it needs to write to X, and immediately after T3 requests a shared lock since it needs to read from X. This order creates a conflict on the locks.

Schedule 1:



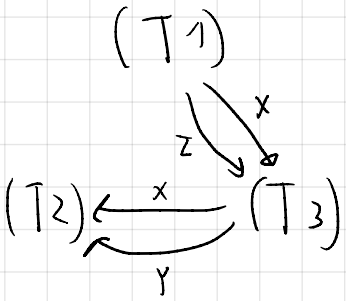
Non-serializable ✗

Schedule 2:



serializable ✓

Schedule 3:



serializable ✓

schedule 1

T1 S(x) R(x) X(Y) W(Y) C

T2 X(Z) W(Z) X(R) W(R) C

T3 S(Z) R(Z) S(y) R(Y) C

schedule 2

T1 S(x) R(x) X(Y) W(Y) C

T2 S(Z) R(Z) X(X) W(X) X(Y) W(Y) C

T3 X(Z) W(Z) C

4

To determine whether $T3$ is allowed to commit or not, the following validation conditions is used: For every pair of transactions Ti and Tj s.t. $TS(Ti) < TS(Tj)$, one of the following *validation conditions* must hold:

- Ti completes (all three phases) before Tj begins.
- Ti completes before Tj starts its WRITE phase, and Ti does not write any database object read by Tj.
- Ti completes its READ phase before Tj completes its READ phase, and Ti does not write any database object that is either read or written by Tj.

Scenario 1

In the following scenario:

```
T1:  RS(T1) = {1, 2, 3}, WS(T1) = {3},
T1 completes before T3 starts.
T2:  RS(T2) = {2, 3, 4}, WS(T2) = {4, 5},
T2 completes before T3 begins with its write phase.
T3:  RS(T3) = {3, 4, 6}, WS(T3) = {3},
```

For $T1$ and $T3$ the first validation condition holds, since $T1$ completes all three phases before $T3$ starts. For $T2$ and $T3$ the first validation does not hold. The second validation condition does not hold either, as though $T2$ completes before $T3$ begins its WRITE phase, $T2$ writes database objects read by $T3$, the object being $\{4\}$; this also means that the third condition does not hold, and in this scenario $T3$ is not allowed to commit.

Scenario 2

In the following scenario:

```
T1:  RS(T1) = {1, 2, 3}, WS(T1) = {3},
T1 completes before T3 begins with its write phase.
T2:  RS(T2) = {5, 6, 7}, WS(T2) = {8},
T2 completes read phase before T3 does.
T3:  RS(T3) = {3, 4, 5, 6, 7}, WS(T3) = {3},
```

For $T1$ and $T3$ the first condition applies, and so does the second and third. For $T2$ and $T3$ the first condition does not apply. $T2$ completes its READ phase before $T3$ completes its READ phase, and $T2$ does not write any database object that is either read or written by $T3$. Thus the third condition holds, and in this case it is given that the second condition also holds. $T3$ is thus allowed to commit.

Scenario 3

In the following scenario:

```
T1:  RS(T1) = {2, 3, 4, 5}, WS(T1) = {4},
T1 completes before T3 begins with its write phase.
T2:  RS(T2) = {6, 7, 8}, WS(T2) = {6},
```

```
T2 completes before T3 begins with its write phase.
T3:  RS(T3) = {2, 3, 5, 7, 8}, WS(T3) = {7, 8},
```

For neither $T1$ and $T2$ the first condition holds. Both $T1$ and $T2$ completes before $T3$ starts its WRITE phase, and neither $T1$ nor $T2$ write any database object read by $T3$, thus the second condition holds. $T3$ is allowed to commit.