ACS - Assignment 2

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

December 3, 2024

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 challenges. One of these challenges is overhead, as the sub-tasks might not be executed ideally in parallel, so some must be done sequentially. The need for communication between the sub-tasks is also created, and some sub-tasks might need access to the same resources. Furthermore if only some part of a program is parallelized, but the rest is not equipped to handle this implementation, the system might be forced to wait for the slowest of the subtasks. Thus in practice this introduce challenges such as race conditions and deadlocks[1].

1.2

Batching is the act of performing several of the same operations one time instead of doing them one at a time[2]. *Dallying* delays some requests for some amount of time. This is either in the hope that some other request will come and make the first request redundant or else it opens up the opportunity for batching[3]. An example of this could be a storage systems where a block receives multiple writes. Dallying would then mean that some request got increased latency in order to wait the next write to the same block. Instead of sending the first write and then the next write, the writes would the be send together, i.e. Batching, so the first write woud not be performed at all, because the second write would update the same block, making the previous write unnecessary.

1.3

When a web browser maintains 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

[1] Compendium chapter 6.1.4 "Reducing Latency using Concurrency"
[2] Compendium chapter 6.1.7.1 "Batching"
[3] Compendium chapter 6.1.7.2 "Dallying"

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 having a storage system as a single address space with N entries. We note that N is significantly larger than K. We also assume that the K is finite, not changing. For the abstractions mapping, we think that some sort of hashing would make sense.

The method we think would make sense for the mapping is consistent Hashing. This method runs in constant time which is one of the requirements. In consistent hashing we have a circle. This circle will go from 0 to N-1 where it will also connect. Each of the machines will get a section of the circle. The smart thing about this set-up with consistent hashing is that if one of our machines were to fail we don't have to remap everything, we only have to remap a section of it. This would be the level of fault tolerance that there would be.

There would be a machine that handles the requests from the different machines. Each machine would make their read and write request and the centralized machine would then find the location for the request and give that information back. This machine should be able to handle multiple request at once. 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 do. So a disadvantage without a fail-safe is that this machine handling the request is a single point failure.

With the consistent hashing, each machine will have its section in the circle making sure that there will be no collisions. This then means that multiple machines can 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. The centralized component here is the machine handling the request. This machine also coordinates the order of requests if many requests are coming for the same place.

2.2

```
M1 = Machine responible for handling request.
M2 Makes READ request
M1 finds the request location in the hashing scheme and send it back to M2.
M2 recieves the data from the READ Request
M3 makes a WRITE request
M1 Find the request location in the hashing scheme and writes makes the Write Request
```

So the READ and WRITE operations would be the individual machines making requests. 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. For our solution it should have some atomic behavior in the sense of concurrency. This mean that the machine that handles the request and does the read and write operations should be able to handle multiple. If multiple write request was all directed at the same place then the concur ency would have to stop and the operations should then be done in the correct order, thereby affect the atomicity of the operations.

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 addition of a new key would also be possible and the resizing of the reactions of keys may be impacted here but possible. The faults that can be tolerated are still the handling of machines crashing and removing them from the hash circle. So long as the machine crashing is not the machine handling the requests.

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. This creates a cycle in the graph. As no schedule is conflict-serializable if its precedence graph has cycles, schedule 1 is not conflict-serializable. If we changed the $R(Y)$ from T3 to a $W(Y)$ we would remove the conflict between T2 and T3. This would remove the cycle in the graph and the schedule would be conflict-serializable.
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.

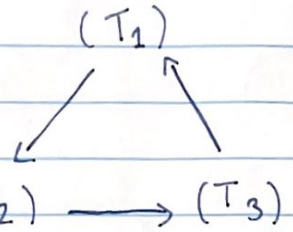
3.2

Schedules 1 could not have been generated by a strict 2PL since it has cycles and is therefore not conflict serializable.

Schedule 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. See schedule on next page.

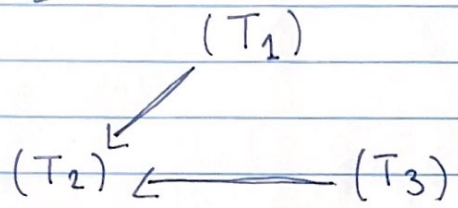
Schedule 3 could not have been generated by strict 2PL, and this can be seen in the very first two actions performed by transaction 1 and 3: T1 acquires an exclusive lock since it needs to write to X, and immediately after T3 requests a shared lock since it needs to read from X. This order creates a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

Schedule 1:



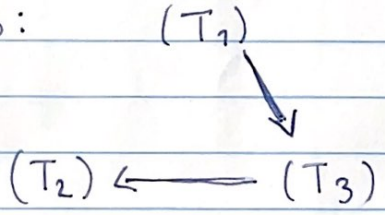
NON-serializable

Schedule 2:



Seridizable

Schedule 3:



serializable

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},
```

T1 does not complete its phases before T3 so the first condition is not met. T1 does complete all its phases before T3 starts its WRITE phases but T1 writes to an object that is in T3's database, $\{3\}$, therefore the second condition is not met either. It does not say explicitly that T1 is done with its read before T3, but even if it is T1 still writes to $\{3\}$ which is in T3 database and therefore does not uphold the third condition either. So none of the conditions is met here and in this scenario, T3 is not 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.