8-hour Online Written Exam of Advanced Computer Systems – ACS, 2020-2021

Department of Computer Science, University of Copenhagen (DIKU)
Date: January 22nd, 2021

Preamble

This is your final 8-hour online written exam for Advanced Computer Systems, block 2, 2020/2021. The exam will be evaluated on the 7-point grading scale with external grading, as announced in the course description.

- Hand in a **single PDF file**, in which your exam number is written on every page.
- Your answers must be provided in English.
- Hand-ins for this exam must be individual, so **cooperation with others in preparing a solution is strictly forbidden.**
- The exam is open-book and all aids are allowed. If you use **any sources other than book or reading material for the course**, they **must be cited appropriately.**
- You may also use your computer or other devices, but you may not access online resources or communicate with any other person in preparing a solution to the exam.
- Remember to write your exam number on all pages.
- You can exclude this preamble in your hand-ins if you directly write on this file.
- This exam has a total of 18 pages excluding this preamble, so please double-check that you have been given a complete exam set before getting started.

Expectations regarding formatted spaces

For each question in this exam, you will find formatted space where you can provide your answer. The spaces provided are designed to be large enough to provide satisfactory (i.e., concise and precise) answers to each of the questions.

Expectations regarding question importance

Questions carry indicative weights. The weights will be used during evaluation to prioritize question answers towards grading; however, recall that the exam is still evaluated as a whole. In other words, we provide weights only as an indication so that you can prioritize your time, should you need to during the exam. You cannot assume that weights will be divided equally among subquestions.

ACS Exam 2021

Exam number 307

2020-01-22

1 Atomicity

- *Give three flavors of atomicity.*

We have discussed many flavors, but three of them that I can recall are *all or nothing atomicity*, *before-or-after atomicity*, and *durability atomicity*.

All-or-nothing atomicity aims to ensure that from the point of view of the database, any given transaction is always either finished or not at all initiated, which is important since half-baked results can disturb other transactions. This can be hard to achieve since if transactions depend on each other (eg. if they access the same memory), then ensuring the property holds for one transaction is dependent on ensuring the property for other transaction, especially in the case of failure. One way of ensuring this flavor of atomicity is logging changes before the transactions commits such that they can be undone.

Before-or-after atomicity is the idea of scheduling and executing transactions such that from the point of view of the invoker, the actions occur either completely before or completely after one another. This can be hard to achieve since it involves carefully analyzing the access patterns of transactions and devising a serializable schedule - this can be done with eg. two-phase locking. However, sometimes it is simply impossible to interleave transactions and still achieve a serializable schedule - in this case the only way to guarantee before-or-after atomicity is to literally run them sequentially.

Durability atomicity is discussed with respect to crash recovery. It is the notion of storing objects in permanent stable storage such that they can always be access at any later point. This is hard to achieve since in practice there is no such thing as permanent storage - however, with eg. redundancy (RAID), stable storage can be *approximated* (meaning it is practically indefinite).

---

2 Performance

2.1 Bottlenecks

- *What are the performance bottlenecks in the two scenarios?*

One bottleneck that the first scenario exhibits is the case where the aggregation stage of the computation is as large as or larger than the computation stage. Consider for example squaring a list of numbers and then summing the results; the squaring would be assigned to the workers, while the summation would necessarily be done by the aggregator. Here, the longest path of execution is $O(n)$ long, since that is how long it takes the aggregator to sum n numbers, and then the benefits of parallelization are possibly lost in the overhead of communication.

The given example only holds for scenario 1, but the same problem is exhibited by scenario 2. A solution to this problem is have multiple aggregation steps, which can potentially reduce the time of aggregation from being *linear* in the size of input to being *logarithmic* in the size of the input. This is known as a reduction tree, and is largely what the architecture behind MapReduce is based on, so using MapReduce would also be a good solution.

Another bottleneck that impedes both systems is data skew. Consider the histogram example given in the text - if each worker is eg. given one file of text each, then one file may be 3 words long while another may be a million words long. The performance of such a system is bounded by the largest file, which may overshadow even the aggregation step.

This type of problem is actually the fault of the scheduler, and the solution to this is to have more clever data partitioning, such that all workers are given equal workload.

2.2 Performance Evaluation

- *How would you evaluate the performance of the above architecture in both scenarios?*

First off I would want to compare the execution time with that of a single worker thread, in order to examine the overhead in parallelization. In this regard it would also be beneficial to evaluate the aggregation step in isolation to figure out where the bottleneck lies in the system.

Wrt. performance metrics, I would first and foremost consider throughput. For most operations this is expected to increase as the number of workers increases, while for others the overhead of communication as well as the aggregation step might be dominant.

Then, I would measure the amount of extra resources required for the parallelization and weigh it against the improve in throughput to determine whether parallelization is worth the cost.

- *Would you setup the experiments differently for the two scenarios?*

No. I would perform the same experiments on both scenarios to better be able to compare the results, and in order to better determine which scenario the architecture handles better.

3 Concurrency Control

3.1 Schedules

- **Schedules, 3.1.1:** *Give a serializable schedule of T1, T2, and T3.*

To paraphrase from the compendium, a schedule S_1 is serializable if, on any consistent database, it can be rewritten into another S_2 schedule which is totally serial.

While some definitions of serializability do not require that S_1 is not in itself a totally serial schedule, the course compendium is not clear on this, so let's have all three transactions execute serially but interleave their commits:

```
1 T1: R(X) W(X) R(Y) W(Y)                                C
2 T2:                  R(Y) W(Y) W(Z)                 C
3 T3:                                R(X) R(Z) W(Z) C
```

- **Schedules, 3.1.2:** *Give a conflict-serializable schedule of T1, T2, and T3.*

A schedule is conflict-serializable if its precedence graph is **aclycic**. Let's again consider the schedule given for task **3.1.1** above:

```
1 T1: R(X) W(X) R(Y) W(Y)                                C
2 T2:                  R(Y) W(Y) W(Z)                 C
3 T3:                                R(X) R(Z) W(Z) C
```

This schedule is conflict-serializable. To show this, we draw the precedence graph, which is a graph whose vertices are the three transactions, and which contains an edge $T_i \rightarrow T_j$ whenever an action of T_i precedes and conflicts with one of T_j's actions. The graph looks like so:

$$V = \{T1\,, T2\,, T3\}$$
$$E = \{T1 \rightarrow T2\,, T2 \rightarrow T3\,, T1 \rightarrow T3\}$$

We see that there are no cycles in the precedence graph, and conclude that the schedule is conflict-serializable.

- **Schedules, 3.1.3:** *Give a schedule of T1, T2, and T3 that is **not** conflict-serializable.*

To give a schedule that is not conflict-serializable, we simply need to insert a cycle in the precedence graph. The below schedule is identical to the previous, except T1's write to Y is moved after T2's write to Y:

```
1 T1: R(X) W(X) R(Y)           W(Y)                      C
2 T2:                R(Y) W(Y)      W(Z)                C
3 T3:                                    R(X) R(Z) W(Z) C
```

The precedence graph now also contains an edge $T2 \rightarrow T1$, meaning there is a cycle $T1 \rightarrow T2 \rightarrow T1$.

- **Schedules, 3.1.4:** *Give a conflict-serializable schedule of T1, T2, and T3 that could **not** have been generated by a strict 2PL scheduler with lock upgrades.*

To give an example of this, let's consider the completely serial schedule, but move T3's read of X after T1's write to X, as below:

```
1 T1: R(X) W(X)      R(Y) W(Y) C
2 T2:                           R(Y) W(Y) W(Z) C
3 T3:           R(X)                              R(Z) W(Z) C
```

This schedule is conflict serializable since its precedence graph:

$$V = \{T1\,, T2\,, T3\}$$
$$E = \{T1 \rightarrow T2\,, T2 \rightarrow T3\,, T1 \rightarrow T3\}$$

is still aclycic, but is could **not** have been generated by a strict 2PL scheduler because when T3 wants to read X, it needs at least a shared lock of X, but it cannot obtain this since T1 holds an exclusive lock, which it cannot let go of before it commits.

- **Schedules, 3.1.5:** *Give a conflict-serializable schedule of T1, T2, and T3 that could have been generated by a strict 2PL scheduler with lock upgrades, but not by a conservative 2PL scheduler with lock downgrades.*

I give this example:

```
1 T1: R(X) W(X) R(Y) W(Y) C
2 T2:                       R(Y) W(Y)                 W(Z) C
3 T3:                                  R(X) R(Z) W(Z) C
```

The schedule has the following precedence graph:

$$V = \{T1\,, T2\,, T3\}$$
$$E = \{T1 \rightarrow T2\,, T1 \rightarrow T3\,, T3 \rightarrow T2\}$$

which is aclycic, so the schedule is conflict-serializable.

The schedule could have been produced by a S2PL scheduler, since T3 does not need the lock on Y, and because T3 releases the exclusive lock on Z before T2 needs it.

However, it could not have been produced by a C2PL scheduler, since T2 cannot perform its read and write of Y before it has also acquired the lock on Z, which it does not obtain until T3 has released it.

3.2 Deadlocks

- **Deadlocks, 3.2.1:** *Which of the 2PL variants can result in deadlocks? Explain why and give a schedule with a deadlock. Explain why other variants cannot generate deadlocks.*

Strict two-phase locking can produce deadlocks, since it begins execution before it has acquired all locks, and does not let go of any of them before it commits. If two transactions begins execution before they have acquired all necessary locks, they might end up in a situation where they need a lock held by the other, but neither of them can release the lock needed by the other transaction.

Below schedule is an example of this, where `S(A)` and `X(A)` means shared and exclusive locks of `A`:

```
1 T1:           X(C)           X(A)
2 T2:      X(B)           X(C)
3 T3: S(A)           S(B)
```

Here, T3 waits for T2's lock on B; T2 waits for T1's lock on C; and T1 waits for T3's lock on A.

Other 2PL variants which gather all locks before they start execution cannot produce deadlocks, since this would imply that they will never be in a situation where they are both holding locks and waiting for other locks.

- **Deadlocks, 3.2.2:** *Which of the two strategies prevent deadlocks? For those that do, explain why, and else give a counterexample schedule.*

Both strategies succeed. Since locks are atomic, only one transaction can request a given lock at a time, meaning only that transaction can be responsible for a deadlock at a time. Removing one node from a cycle is sufficient to break that cycle and thus the deadlock.

4 Recovery

4.1 The Missing Log

- *Complete the log given in the exam text.*

Since my exam number is 307, the second page table entry is `P307` and its `recLSN` is 6. Below is the log:

```
LSN | PREV_LSN | XACT_ID | TYPE       | PAGE_ID | UNDONEXTLSN
----+----------+---------+------------+---------+------------
1   | -        | -       | begin CKPT | -       | -
2   | -        | -       | end CKPT   | -       | -
3   | NULL     | T3      | update     | P42     | -
4   | 3        | T3      | commit     | -       | 3
5   | NULL     | T1      | update     | P99     | -
6   | NULL     | T2      | update     | P307    | -
7   | 6        | T2      | abort      | -       | 6
8   | 5        | T1      | update     | P99     | 5
+++ CRASH +++ CRASH +++ CRASH +++ CRASH +++ CRASH +++ CRASH +++
```

4.2 The Recovery

- *Based on the log, complete the recovery procedure. Show the 5 items mentioned in the exam text.*

Set of winners and losers

The set of winners is $\{T3\}$, since T3 was the only transaction to commit before the crash. The set of losers is thus $\{T2, T1\}$.

LSNs for start of redo and end of undo

The redo phase starts at LSN 3, since this is the smallest recLSN in the DPT.

Since initially, toUndo $= \{7, 8\}$, we will undo the transactions in the order 8, 7, 6, 5. The undo phase thus ends at LSN = 5.

Set of log records that may cause pages to be rewritten during redo

The set of log records that may cause pages to be rewritten during redo is $\{3\,, 6\,, 8\}$. *Notice that this set* ***does not*** *contain the update with LSN = 5; this is because its LSN is less than the recLSN for P99 (which is 8).*

Set of log records undone during undo

The set of log records undone during undo is $\{8, 7, 6, 5\}$. This is because intially, we have toUndo $= \{7, 8\}$, and during the undo phase we will we will first undo 8, which adds 6 to toUndo; next, we undo 7, which adds 6 to toUndo; then, we undo 6 and 5, neither of which add anything to toUndo, since their PREV_LSN are NULL.

Contents of the log after recovery completes

LSN	PREV_LSN	XACT_ID	TYPE	PAGE_ID	UNDONEXTLSN
1	-	-	begin CKPT	-	-
2	-	-	end CKPT	-	-
3	NULL	T3	update	P42	-
4	3	T3	commit	-	3
5	NULL	T1	update	P99	-
6	NULL	T2	update	P307	-
7	6	T2	abort	-	6
8	5	T1	update	P99	5
+++ CRASH +++ CRASH +++ CRASH +++ CRASH +++ CRASH +++ CRASH +++					
9	4	T3	end	-	-
10	8	T1	CLR	P99	5
11	7	T2	abort	-	6
12	6	T2	CLR	P307	NULL
13	12	T2	end	-	-
14	5	T1	CLR	P99	NULL
15	14	T1	end	-	-

First, notice that since T3 had committed before the crash, an end record is written for T3 during analysis.

Then, undo commences, undoing log records in the order 8, 7, 6, and finally 5. Since T2 aborted at LSN = 7, this is re-recorded. Log records for T2 and T1 are written after they have been fully undone.

5 Reliabilty

5.1 Lamport and Vector Clock

- **Lamport and Vector Clock, 5.1.1:** *Write the Lamport logical clock of each event.*

```
e1  = 1
e2  = 2
e3  = 2
e4  = 3
e5  = 3
e6  = 3
e7  = 4
e8  = 4
e9  = 5
e10 = 6
e11 = 5
```

- **Lamport and Vector Clock, 5.1.2:** *Write the vector clock of each event.*

```
e1  = (1, 0, 0)
e2  = (1, 1, 0)
e3  = (2, 0, 0)
e4  = (2, 0, 1)
e5  = (3, 1, 0)
e6  = (2, 2, 0)
e7  = (4, 1, 0)
e8  = (2, 0, 2)
e9  = (5, 2, 0)
e10 = (6, 2, 1)
e11 = (4, 1, 3)
```

- **Lamport and Vector Clock, 5.1.3:** *Can P1 tell that the message received at e10 was actually sent before the one received at e9 using either Lamport or vector clock?*

Using Lamport timestamps

If in real time some event e_i happened before another event e_j, then $\text{LC}(e_i) < \text{LC}(e_j)$, where $\text{LC}(e)$ is the Lamport clock of some event e. However, the implication does not go the other way!

Hence, P1 cannot use the Lamport timestamps to tell that $e4$ happened before $e6$, only that $e9$ happened before $e10$.

Using vector timestamps

Yes! P1 can use ector timestamps to tell that $e4$ happened before $e6$.

This is because with vector clocks, the aforementioned implication *does* holds the other way - ie. if $\text{VC}(e_i) < \text{VC}(e_j)$, where $\text{VC}(e)$ is the vector clock for event e, then e_i happened before e_j in real time.

- **Lamport and Vector Clock, 5.1.4:** *What are the correct responses of Shipment to the two NewOrder messages, and how can Shipment enforce these responses?*

Upon receiving the two NewOrder messages for the same OrderID, Shipment should respond that this order has already been cancelled. Shipment can enforce this response by keeping track of cancelled orders, even if those orders have not been seen by Shipment yet (this of course assumes that Order never repeats OrderIDs).

5.2 Replication

- **Replication, 5.2.1:** *Compare pros and cons of synchronous and asynchronous replication for replicating Shipment.*

In the following I will assume that the replicas are only used to backup state and receive messages, and that only the main Shipment server sends back messages, since the exam text does not state otherwise.

Asynchronous replication, of course, has the advantage over synchronous replication in that it requires significantly less bandwidth, which might be useful if replicas reside far away from the main Shipment server. However, synchronous replication has the immense benefit that fail-over is much faster in case of main Shipment server failure.

- **Replication, 5.2.2:** *Would a write quorum of 3 and a read quorum of 2 be problematic for 5 replicas of Shipment?*

$Q_w = 3$ is ideal, since $3 > 5/2$; in other words, only one transaction can write a time.

However, if $Q_w = 3$, then Q_r needs to also be at least 3, since otherwise two transactions can read and write at the same time. So yes, this setup is problematic!

- **Replication, 5.2.3**

Blank.

---

6 Distributed Transaction

6.1 Basic Concepts of Distributed Transactions

- **Basic Concepts of Distributed Transactions, 6.1.1:** *Does the DanskeShop setup fulfill the atomicity property in case of node failures? Use an example to explain why or why not.*

It does *not*, since it only uses a one-phase commit. Consider this example: P1 sends a NewOrder message to both Payment and Shipment. Payment agrees and sends back an AckOrder message to P1. Meanwhile, there is a node failure at Shipment and so the NewOrder message is not even received at Shipment.

Payment does not wait for a doCommit message from Order since that is not part of the protocol, but rather simply begins processing the request as if all is good. However, the order is never shipped, so the customer will be paying for nothing.

6.2 Two-Phase Commit

- **Two-Phase Commit, 6.2.1:** *Describe the 2PC commit procedure of a New Order transaction when there is no failure. Write down the coordinator's log records.*

1. Coordinator writes "Prepare order" to log.
2. Coordinator sends `canCommit?`-messages to all three modules.
3. Coordinator receives `yes` from all three modues.
4. Coordinator writes "Commit order" to log.
5. Coordinator sends `doCommit`-messages to all three modules.
6. Coordinator receives `haveCommited` messages from all three modules.
7. Coordinator writes "End order" to log.

- **Two-Phase Commit, 6.2.2:** *Describe a scenario in which Shipment and Payment can proceed to complete the transaction.*

The exam text does not state what is meant by "to complete the transaction". *I will assume that a transaction is completed if it is either commited or aborted.*

Also, the text says that Order fails after the coordinator decides to abort and after the abort has been logged, but *before* abort messages are sent out.

Then Shipment and Payment can proceed to complete the transaction as such:

1. having already logged the abort, the coordinator sends rollback messages to all three modules. Payment and Shipment receive these.
2. Payment and Shipment each rollback the commits and send back `haveRolledBack`-messages to the coordinator, thus completing the transaction with an abort (as per the assumption that a transaction is completed when it is either committed or aborted).

When Order comes back online, the coordinator can retry the transaction if it wishes.

- **Two-Phase Commit, 6.2.3:** *Under the given scenario, describe what could happent to Shipment if it has not received the abort message, and what could happen to Order after it restarts from the failure.*

When Order restarts from failure, it will look for a commit message in its log. However, it will not find one and instead assume that it must have aborted.

---

7 Data Processing

7.1 Selection

- **Selection, 7.1.1**

Since there is no indexing, there is nothing to do but to use a sequential search algorithm. This costs N I/O's, where N is the number of pages - in this case the Orders table is 25,000 pages large, so the number of I/O's is 25,000.

(The search can of course be parallelized, but we still need to examine all 25,000 pages).

- **Selection, 7.1.2**

When the index is non-clustered, there is a risk of random I/O's since the index may not lie in the same order as the actual records in storage. In the worst case, the number of I/O's can be equal to the actual number of records requested.

This means that in this case the worst case number of I/O's is 50,000, whereas with the sequential scan the worst case number of I/O's is 25,000, since that is the number of pages.

For this reason the sequential scan is *very likely* to be better in this case.

7.2 Join and Aggregate

- **Join and Aggregate, 7.2.1:** *What is the most efficient algorithm to perform the join, and what is its I/O cost?*

In the following I will assume that 1 buffer page can hold 1 page of Orders or Shipments.

The most efficient algorithm is hybrid hash-join. If R and S are relations; $B(R)$ is the number of buffer pages needed to hold R, and we have $B(S) \leq B(R)R$, then the general formula for the number of I/O's for hybrid hash-join is :

$$\left(3 - \frac{2M}{B(S)}\right) \cdot (B(R) + B(S))$$

In our case, we have $M := 150$, and because $B(Shipments) < B(Orders) \equiv 1500 < 25000$, we set $S := Shipments\,, R := Orders$, and the number of I/O's needed for a hybrid hash-join in this case becomes:

$$\left(3 - \frac{2 * 150}{1500}\right) \cdot (25000 + 1500) = 74200$$

This is not a big improvement over regular hash-join, which would need:

$$3 \cdot (25000 + 1500) = 74200$$

79500 I/O's in this case. However, this could be improved with more main memory.

- **Join and Aggregate, 7.2.2:** *If the server could be upgraded with more main memory, is there a better choice of algorithm?*

If main memory can be increased *arbitrarily*, then the best we can do in terms of the number of I/O's is to simply load the smaller of the two relations into main memory once, and then to load the larger relation in as needed. This could eg. be done using block-based nested-loop join, and in this case, the number of I/O's would simply be $B(R) + B(S)$, while the minimum amount of memory needed would be $\min(B(R), B(S))$ for some two relations R and S.

In this case, we have $B(Shipments) = 1500$, so we would need $M \geq 1500$

- **Join and Aggregate, 7.2.3:** *Describe the most efficient algorithm to group and sort the result from the previous step when this contains 100,000 pages and there are now 500 available buffer pages.*

If we know that we need the output sorted, we can use a sorting-based grouping/aggregation operator, which takes $3B(R)$ disk I/O's. This has the benefit that the result is already sorted by productID; however, it requires $B(R) \leq M^2$ memory.

Now that we have beefed up the number of available buffer pages to 500, we have enough main memory, since $100,000 < 500^2 = 250,000$. The number of I/O's needed to perform the group+aggregate+sort is thus: $3 * 100,000 = 300,000$.

- **Join and Aggregate, 7.2.4**

Blank.