CARNEGIE MELLON UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
15-445/645 – DATABASE SYSTEMS (SPRING 2023)
PROF. CHARLIE GARROD

Homework #4 (by Abigale Kim) – Solutions

Due: **Friday April 7, 2023 @ 11:59pm**

IMPORTANT:

- Enter all of your answers into **Gradescope by 11:59pm on Friday April 7, 2023**.
- **Plagiarism**: Homework may be discussed with other students, but all homework is to be completed **individually**.

For your information:

- Graded out of **100** points; **4** questions total
- Rough time estimate: ≈ 2 - 4 hours (0.5 - 1 hours for each question)

Revision : 2023/04/11 22:06

Question	Points	Score
Query Execution, Planning, and Optimization	25	
Serializability and 2PL	29	
Hierarchical Locking	20	
Multi-Version Concurrency Control	26	
Total:	100	

Question 1: Query Execution, Planning, and Optimization . . . . . [25 points]

Graded by:

(a) **[5 points]** The zone map optimization is more useful in speeding up OLAP queries compared to OLTP queries.

[✓ **True**] ☐ False

Solution: Recall that the zone map optimization is pre-computing aggregations for each tuple attribute in a page. This would be more useful for speeding up OLAP queries, since OLAP DBMSs (which store columns of data contiguously) are usually the DBMS of choice when running aggregate queries.

(b) **[5 points]** The thread per worker process model allows for the opportunity of using intra-query parallelism, whereas the process per worker process model does not.

☐ True [✓ **False**]

Solution: This is false, since within the worker process itself is still capable of equipping intra-query parallelism to execute a query. Intra-query parallelism just means that a DBMS executes the operations of a single query in parallel, on multiple processors.

(c) **[5 points]** The vectorized query processing model (which uses SIMD instructions to parallelize operations) is an example of intra-query parallelism.

[✓ **True**] ☐ False

Solution: This is true. Intra-query parallelism is when the DBMS executes the operations of a single query in parallel. Therefore, using SIMD instructions within a vectorized processing model to execute a query would be an example of intra-query parallelism.

(d) **[5 points]** An index scan is always better (fewer I/O operations, faster run-time) than a heap scan if the query contains an `ORDER BY` clause matching the index key.

☐ True [✓ **False**]

Solution: An index scan may require multiple I/O operations per result tuple (to look up the row in the index, then retrieve additional attributes in the heap) whereas a sequential scan will just go through the heap. Moreover, a sequential heap scan may benefit more from pre-fetching pages. PostgreSQL's optimizer defaults to picking sequential scans over index scans if it thinks that you're asking for more than (very approximately) 10% of all rows in the table.

(e) **[5 points]** Database management systems estimate the cost of every plan for a query, to pick the optimal plan for execution.

☐ True [✓ **False**]

Solution: No, it is usually not necessary to estimate the cost of every plan for a query via a cost model. In this case, the time it would take to enumerate every plan and then filter out the plans to pick the most optimal one would introduce too high of an overhead

Question 1 continues. . .

compared to the query time itself. Usually, DBMSs will use rule-based optimizations (or heuristics) first, transforming the plan into a more simple one.

Homework #4 continues…

Question 2: Serializability and 2PL............................[29 points]

(a) True/False Questions:

i. **[2 points]** One downside of regular (i.e., not strong strict) 2PL is that it can lead to a domino effect, where multiple transactions are aborted.
[✓ **True**] ☐ False

Solution: This is true, regular 2PL can lead to cascading aborts. This is because if a transaction unlocks an object after its usage, then its state may be seen by others before it commits. If this transaction aborts, then the others who saw the state of an aborted transaction must also abort.

ii. **[2 points]** Using strong strict 2PL guarantees a conflict serializable schedule.
[✓ **True**] ☐ False

Solution: This is true. Using regular 2PL guarantees a conflict serializable schedule, and a schedule provided by strong strict 2PL has a even stronger guarantee (all the guarantees of using regular 2PL, plus no cascading aborts).

iii. **[2 points]** Using regular (i.e., not strong strict) 2PL guarantees a conflict serializable schedule.
[✓ **True**] ☐ False

Solution: This is true. Using regular 2PL guarantees a conflict serializable schedule because it generates schedules whose precedence graph is acyclic. This is because this ordering of acquiring resources (via the locks) ensures that there is a order of acquisition precedence.

iv. **[2 points]** View serializable schedules never produce unrepeatable reads.
[✓ **True**] ☐ False

Solution: Recall that unrepeatable reads happen when a transaction that reads the same value twice will not get the same value. View serializable are not prone to unrepeatable reads because view serializable schedules are still view equivalent with a serial schedule (where one transaction executes after one another), which means that reading a initial value, then reading a different value written by another transaction should not be possible.

v. **[2 points]** Strong strict 2PL prevents deadlocks during transaction execution.
☐ True [✓ **False**]

Solution: This is false because strong strict 2PL acquires the locks in the same way regular 2PL does, and regular 2PL acquires locks by transaction operation order. In the most simple case, consider two data A and B, and transactions T_1 and T_2 that operate on both A and B. If T_1 locks A first, and T_2 locks B first because it operates on B first, then this causes a deadlock.

vi. **[2 points]** Regular (i.e., not strong strict) 2PL prevents the lost update problem.
[✓ **True**] ☐ False

Question 2 continues...

Solution: This is true. According to the paper that explains lost updates, a lost update happens when a transaction T_1 reads a data item, and then T_2 updates the data item, and then T_1 updates the data item and commits. This technically erases T_2's update. An example schedule without 2PL would look like this.

`R(X)`	
	`W(X)`
`W(X)`	

With 2PL, we would have to grab a lock on X in T_1 to even read it, which would ensure that this schedule does not happen.
Because the staff thinks this was not explained in class adequately, everyone will be getting full credit for this question on the homework.

Grading info: -2 for each incorrect answer

(b) Serializability:
Consider the schedule of 4 transactions in Table 1. R(·) and W(·) stand for 'Read' and 'Write', respectively, and time increases from left to right. (This is in contrast to the diagrams in class, where time proceeded downward.)

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
T_1		W(B)		W(A)					R(C)	
T_2	W(E)							R(A)		
T_3			R(B)							R(D)
T_4					R(D)	R(E)	W(C)			

Table 1: A schedule with 4 transactions

i. **[2 points]** Is this schedule serial?
☐ Yes [✓ **No**]

Solution: This schedule isn't serial because this schedule interleaves the actions of different transactions.

ii. **[2 points]** Is this schedule serializable?
☐ Yes [✓ **No**]

Solution: This schedule isn't serializable because it is not equivalent (in terms of effect on the database) to any serial ordering of the transactions.

iii. **[2 points]** Is this schedule conflict serializable?
☐ Yes [✓ **No**]

Solution: This schedule isn't conflict serializable because there is a cycle in its data dependency graph. Moreover, this schedule is not conflict equivalent (every pair of conflicting operations is ordered in the same way) to any serial schedule of transaction execution.

iv. **[2 points]** Is this schedule view serializable?
☐ Yes [✓ **No**]

Question 2 continues…

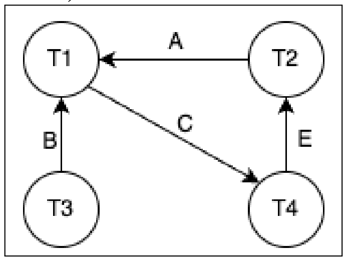
Solution: This schedule isn't view serializable because this schedule is not view equivalent to any serial schedule of transaction execution. Note that if we order the transactions T_1, T_2, and T_4 in any serial manner, we cannot reach the database state of this ordering of actions.

Grading info: -2 for incorrect answer

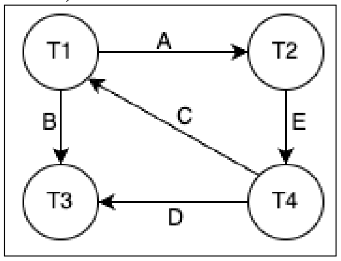
Question 2 continues. . .

v. **[4 points]** Choose the correct dependency graph of the schedule given above. Each edge in the dependency graph looks like this: '$T_x \rightarrow T_y$ with Z on the arrow indicating that there is a conflict on Z where T_x read/wrote on Z before T_y'.

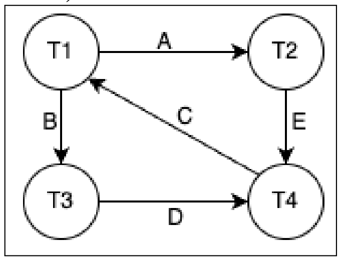
☐ A)



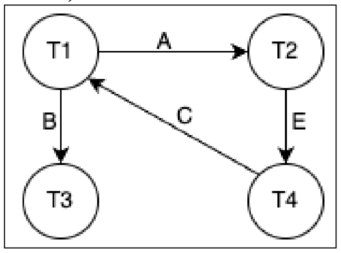
☐ B)



☐ C)



■ **D)**



Solution: D is the correct answer. There are 4 edges in the graph:

- An edge from T_1 to T_2, noting the write of A by T_1 at t_4 and the read of A by T_2 at t_8
- An edge from T_2 to T_4, noting the write of E by T_2 at t_1 and the read of E by T_4 at t_6
- An edge from T_4 to T_1, noting the write of C by T_4 at t_7 and the read of C by T_1 at t_9
- An edge from T_1 to T_3, noting the write of B by T_1 at t_2 and the read of B by T_3 at t_3

There should be no other edges in the graph.

Grading info: -4 for incorrect answer.

vi. **[3 points]** Select all possible changes that would produce a conflict serializable schedule.

■ **Remove only T1**
■ **Remove only T2**
☐ Remove only T3
■ **Remove only T4**
☐ Original schedule is already conflict serializable, remove no transactions

Solution: Since there is a cycle in the dependency graph between `T1`, `T2`, and `T4`, this indicates that there is a conflict in transaction precedence, and if we remove one of these transactions, the schedule becomes conflict serializable.

Grading info: -3 for incorrect answer

vii. **[2 points]** Is this schedule possible under regular 2PL?

☐ Yes

[✓ **No**]

Solution: This schedule is not possible under 2PL because it is not conflict serializable, and 2PL is guaranteed to produce conflict serializable schedules.

Grading info: -2 for incorrect answer

Homework #4 continues…

Question 3: Hierarchical Locking .... [20 points]

Consider a database `D` consisting of two tables `A` (which stores information about musical artists) and `R` (which stores information about the artists' releases). Specifically:

- `R(rid, name, artist_credit, language, status, genre, year, number_sold)`
- `A(id, name, type, area, gender, begin_date_year)`

Table `R` spans 1000 pages, which we denote `R1` to `R1000`. Table `A` spans 50 pages, which we denote `A1` to `A50`. Each page contains 100 records. We use the notation `R3.20` to denote the twentieth record on the third page of table `R`. There are no indexes on these tables.

Suppose the database supports shared and exclusive hierarchical intention locks (`S`, `X`, `IS`, `IX` and `SIX`) at four levels of granularity: database-level (`D`), table-level (`R` and `A`), page-level (e.g., `R10`), and record-level (e.g., `R10.42`). We use the notation `IS(D)` to mean a shared database-level intention lock, and `X(A2.20-A3.80)` to mean a set of exclusive locks on the records from the 20th record on the second page to the 80th record on the third page of table `A`.

For each of the following operations below, what sequence of lock requests should be generated to maximize the potential for concurrency while guaranteeing correctness?

(a) **[4 points]** Fetch the record where artist `name = 'Taylor Swift'`.

☐ `IX(D), X(A)`
☐ `S(D)`
☐ `IS(D), IS(A)`
■ `IS(D), S(A)`

Solution: The correct choice is `IS(D), S(A)`. This choice is correct because it accesses intended shared locks for all parent levels necessary, and then accesses a shared lock on the DBMS, which is what the DBMS needs to run this query.

- `IX(D), X(A)` is wrong because it accesses an intended exclusive lock, which the DBMS does not need for a read query
- `S(D)` is wrong because it does not get the intended parent locks necessary to get the `S(D)` lock
- `IS(D), IS(A)` is wrong because it does not get the `S(A)` lock for the DBMS. It states only the intention to read-access the relation `A`.

Grading info: -4 for incorrect answer

(b) **[4 points]** Modify the 15th record on `R645`.

☐ `IX(D), IX(R), IX(R645), IX(R645.15)`
■ `IX(D), IX(R), IX(R645), X(R645.15)`
☐ `SIX(D), SIX(R), SIX(R645), IX(R645.15)`
☐ `IS(D), IS(R), IS(R645), X(R645.15)`

Question 3 continues…

Solution: The correct choice is `IX(D), IX(R), IX(R645), X(R645.15)`. This choice is correct because it accesses all intended exclusive locks for all parent levels necessary, and then accesses the exclusive lock for the particular record.

- `IX(D), IX(R), IX(R645), IX(R645.15)` is incorrect because it only gets the intention exclusive lock for the record
- `SIX(D), SIX(R), SIX(R645), IX(R645.15)` is incorrect because the DBMS only intends to write to a tuple, not read any data. Therefore it should not grab the shared-exclusive intention locks.
- `IS(D), IS(R), IS(R645), X(R645.15)` is incorrect because the DBMS intends to write to a tuple, so it should not grab the shared intention parent locks.

Grading info: -4 for incorrect answer

(c) **[4 points]** Scan all the records on pages `A41` through `A49`, and modify `A32.75`.

☐ `IS(D), IS(A), S(A41-A49), IX(A32), IX(A32.75)`
☐ `IS(D), IS(A), S(A41-A49), IX(A32), X(A32.75)`
■ `IX(D), SIX(A), IX(A32), X(A32.75)`
☐ `SIX(D), SIX(A), IX(A32), X(A32.75)`

Solution: The correct choice is `IX(D), SIX(A), IX(A32), X(A32.75)`. This choice is correct because it accesses all intended exclusive locks and the exclusive lock `X(A32.75)`. It also gains a shared lock on `A`, so it can read pages `A41` through `A49`.

- `IS(D), IS(A), S(A41-A49), IX(A32), IX(A32.75)` is incorrect because it accesses an intended shared parent lock for both `D` and `A`, when we plan to modify a record in `A`.
- `IS(D), IS(A), S(A41-A49), IX(A32), X(A32.75)` is incorrect for the same reason as the first choice.
- `SIX(D), SIX(A), IX(A32), X(A32.75)` is incorrect because we do not plan on reading the database, only relation `A`, so we should grab only an intention lock at the database level.

Grading info: -4 for incorrect answer

(d) **[4 points]** Scan all records in `R` and modify the 23^{rd} record on `R7`.

■ `IX(D), SIX(R), IX(R7), X(R7.23)`
☐ `IS(D), IS(R), IS(R7), X(R7.23)`
☐ `S(D), IX(R), X(R7.23)`
☐ `IS(D), SIX(R), X(R7.23)`

Solution: The correct choice is `IX(D), SIX(R), IX(R7), X(R7.23)`. This choice is correct because it accesses all intended exclusive locks and the exclusive lock `X(R7.23)`. It also gains a shared lock on `R`, so it can scan all the records in `R`.

Question 3 continues…

- `IS(D), IS(R), IS(R7), X(R7.23)` is incorrect because we should not gain intended shared locks on —R— if we plan to modify a record in `R`.
- `S(D), IX(R), X(R7.23)` is incorrect because we do not have the prerequsities to gain a `IX` lock on `R` with the shared lock on database `D`. We needed an intended exclusive lock on `D` instead.
- `IS(D), SIX(R), X(R7.23)` is incorrect because we do not have the prerequsities to gain a `SIX` lock on `R` with the intended shared lock on database `D`. We needed an intended exclusive lock on `D` instead.

Grading info: -4 for incorrect answer

(e) **[4 points]** Delete records in `R` if `number_sold < 3`.

☐ `SIX(D), S(R)`
☐ `SIX(D), X(R)`
[✓ `IX(D), X(R)`]
☐ `IX(D), SIX(R)`

Solution: The correct answer choice is `IX(D), X(R)`. This is because we potentially need to modify all the records in `R`, and this choice accesses all intended exclusive parent locks to get the exclusive lock on `R`.

- `SIX(D), S(R)` is incorrect because it gains a shared lock on `R` when it could potentially modify the contents of `R` by deleting entries.
- `SIX(D), X(R)` is incorrect because it gains a shared lock on the database `D` when it has no need to read the contents of `D`.
- `IX(D), SIX(R)` is incorrect because it does not gain the exclusive lock on `R`, which it needs to delete records in `R`.

Grading info: -4 for incorrect answer

Homework #4 continues…

Question 4: Multi-Version Concurrency Control . . . . [26 points]

Graded by:

Scout is a database management system designer who has decided to implement multi-versioning for her system. She is now contemplating design decisions given the current workloads she wants to run, given the following constraints:

1. Most of the data needed in this database has already been loaded.
2. The DBMS is optimized for the speed of highly-concurrent reads.
3. I/O is expensive on this DBMS platform. The DBMS should minimize number of I/O requests per read.
4. Reads often access snapshots of the database taken at different times.
5. The database contains secondary indexes to help speed up read performance.

For each of the design decisions below, provide a brief response (∼3 sentences) that explains which approach Scout should use, along with the pros and cons of that approach.

(a) **[6 points]** Concurrency Control (Two-Phase Locking, Timestamp Ordering, or Optimistic Concurrency Control)

Solution: You can make an argument for either two-phase locking or optimistic concurrency control. Given the constraints, timestamp ordering is incorrect. For each read, the system would introduce a timestamp, and the validation of a serial schedule with timestamp ordering has high overhead. Therefore, this is suboptimal compared to the other two options.

For two-phase locking, here are the pros and cons of the approach:
Pros: It's optimized for high concurrency since with most of the transactions being reads, they will just grab shared locks and will not conflict with one another.
Cons: There's a small chance of deadlock if there happens to be a write or update transaction.

For optimistic concurrency control, here are the pros and cons of the approach:
Pros: Since most of the transactions are reads, there won't be many conflicts, and the system can enjoy the benefit of no lock management.
Cons: There is a high overhead, since OCC copies data locally.

(b) **[6 points]** Version Storage (Append-Only Storage, Time-Travel Storage, or Delta Storage)

Solution: The main thing to realize is that all the storage layouts store version chain pointers. Time travel storage / delta storage has a slight advantage when you access the most updated tuples in the main table, but since we explicitly note that we're accessing snapshots taken at various times, we can definitely work around this.

For append only N2O storage, here are the pros and cons of the approach:
Pros: Given that we are accessing snapshots of the database taken at different times, append-only storage is likely to mix different versions of the same tuple in the same

page, and therefore makes it more efficient when iterating the version chain.
Cons: All versions are mixed in the same table, which makes garbage collection hard. We can only delete a page when it does not contain tuples visible to the user. Since different versions of the tuples are mixed together, we will probably have to move tuples around before we can actually free some space.

For time-travel storage, here are the pros and cons of the approach:
Pros: Accesses are faster for most-recently updated tuple since we just access the main table to fetch the latest version of tuple in a single place. This is likely to reduce the number of I/O requests because main table is small compared with the time-travel table, therefore most of the read requests can go directly to the buffer pool.
Cons: Extra time is required on update to copy tuples to the time travel table.

For delta storage, here are the pros and cons of the approach:
Pros: Since we only store deltas for old versions, we can fetch more information within one page fetching. This is likely to reduce the number of I/O requests when iterating the version chain.
Cons: Extra time is required on constructing the old version of tuple by applying delta.

(c) **[6 points]** Garbage Collection (Background Vacuuming, Cooperative Cleaning, or Transaction-level GC)

Solution: The answer here is either background vacuuming or transaction level GC. Cooperative cleaning is incorrect because when threads check for reclaimable memory when they are traversing the free list, this introduces extra unnecessary computational overhead. Note that this is more work than the collection of the read/write set in transaction level GC because we typically access older versions, and so for every version we iterate through, and for every read that we execute, we have to check for reclaimable memory.

For background vacuuming, here are the pros and cons of the approach:
Pros: This approach optimizes for transaction performance, since transactions can just do the necessary reads and finish running faster than other approaches.
Cons: There is the potential of unnecessary work, as background threads that are checking for versions to garbage collect will not find many versions to remove, as most of the database state is updated.

For transaction level GC, here are the pros and cons of the approach:
Pros: Scanning the entire table takes a significant amount of I/O operations, which are expensive in this system.
Cons: Transaction level GC can introduce slight but extra overhead, since every transaction has to collect its own read/write set and pass it to a garbage collector thread.

Fun fact: most of the well-known DBMSs (Oracle, Postgres, MySQL-InnoDB, Singlestore) use background vacuuming as their GC method.

(d) **[6 points]** Index Management (Logical Pointers or Physical Pointers)

Question 4 continues. . .

Solution: The answer is to have physical pointers. Logical pointers introduce one abstraction layer of additional overhead when reading queries, and the goal of having indexes in our system is to speed up read queries.

For physical pointers, here are the pros and cons of the approach:
Pros: Reads are faster, since it requires only accessing the physical address of the version chain head.
Cons: There is additional overhead when executing an update, as you have to update every index as well with the appropriate physical pointer.

(e) **[2 points]** In a DBMS with MVCC implemented, read-only transactions can block write transactions.
☐ True [✓ **False**]

Solution: Read-only transactions cannot block write only transactions in a DBMS with MVCC implemented because they can read a consistent snapshot of the database with a certain timestamp without acquiring locks, while writes will update the most current version of the DBMS.

Aside
If you are interested in the evaluation of MVCC design decisions, the research paper (link: http://www.vldb.org/pvldb/vol10/p781-Wu.pdf), written by researchers at the National University of Singapore and Carnegie Mellon University (including Andy Pavlo) may be of interest to you.