Assignment 4 Solutions

Wiam Skakri

November 1, 2025

1 Question 1

1.1 Question 1(a)

```
CREATE VIEW Deidentified_Exp AS
SELECT E.expID, AM.modelName, P.age AS patientAge, P.diagnosis,
    E.predictedRisk, E.trueOutcome, E.accuracy
FROM Experiment AS E
JOIN Patient AS P ON E.patientID = P.patientID
JOIN AIModel AS AM ON E.modelID = AM.modelID;
```

1.2 Question 1(b)

```
CREATE VIEW High_Performance_models AS
SELECT E.expID, AM.modelName, E.accuracy, E.predictedRisk, E.runtimeSec
FROM Experiment AS E
JOIN AIModel AS AM ON AM.modelID = E.modelID
WHERE E.accuracy >= 0.9 AND E.runtimeSec < 100;
```

1.3 Question 1(c)

```
CREATE VIEW Model_Avg_Accuracy AS
SELECT AM.modelID, AM.modelName, AVG(E.accuracy) AS avgAccuracy
FROM AIModel AS AM
JOIN Experiment AS E ON AM.modelID = E.modelID
GROUP BY AM.modelID, AM.modelName;
```

2 Question 2

Q1 (5): Find the expID and model names where the model accuracy ≥ 0.95.

Answer: `High_Performance_models` (view b)

This view contains expID, modelName, and accuracy. We can query it with an additional filter for accuracy ≥ 0.95.

Q2 (5): List the model names and their accuracy with predictedRisk < 0.3.

Answer: `Deidentified_Exp` (view a)

This view contains modelName, accuracy, and predictedRisk, so we can filter by predictedRisk < 0.3.

Q3 (5): List all patient age and diagnosis, along with predictedRisk and accuracy.

Answer: `Deidentified_Exp` (view a)

This view contains patientAge (aliased from age), diagnosis, predictedRisk, and accuracy. It includes all the required attributes.

Q4 (5): List each AI model and its average accuracy across all experiments.

Answer: `Model_Avg_Accuracy` (view c)

This view is specifically designed for this query. It already contains modelID, modelName, and avgAccuracy (pre-aggregated across all experiments), so it can be queried directly without additional aggregation.

Q5 (6): List all expID of the experiments with model accuracy ≥ 0.90 and the domain of the models in "medical".

Answer: None of the views

Neither `Deidentified_Exp` nor `High_Performance_models` contains the `domain` column from the AIModel table, which is required to filter by domain = "medical". Therefore, we must query the base tables.

3 Question 3

3.1 Scenario A

ACID Property: Atomicity

Explanation:

This scenario violates the **Atomicity** property of ACID. Atomicity requires that a transaction must be treated as a single, indivisible unit of work—either all operations within the transaction are completed successfully, or none of them are applied to the database.

In Scenario A, the AI agent crashes after reducing the available seats in Trip.availableSeats but before inserting the booking record into the Booking table. This represents a partial execution of the transaction. After recovery, the system restored to a state where no seats were deducted and no booking exists, which indicates that the transaction was rolled back.

3.2 Scenario B

ACID Property: Durability

Explanation:

This scenario demonstrates the **Durability** property of ACID. Durability ensures that once a transaction has been committed, its effects are permanent and will persist even in the face of system failures such as crashes, power outages, or hardware failures.

In Scenario B, the AI agent successfully completes the booking transaction, which means the transaction was committed to the database. Immediately after the commit, the system crashes. However, when the system restarts, both the booking record and the seat deduction are still present in the database.

This shows that durability is being properly maintained. The database management system ensured that the committed transaction's changes were written to permanent storage (disk) rather than just being kept in volatile memory. As a result, the changes survived the system crash and remained persistent after recovery. This is exactly what the durability property guarantees: once a transaction is committed, its results are permanently recorded in the database, regardless of subsequent system failures.

3.3 Scenario C

ACID Property: Isolation

Explanation:

This scenario violates the **Isolation** property of ACID. Isolation ensures that concurrent transactions do not interfere with each other and that each transaction executes as if it were the only transaction running on the database. The intermediate states and operations of one transaction should not be visible to or affect other concurrent transactions.

In Scenario C, two AI agents simultaneously attempt to book the last seat on a trip. Both agents read availableSeats = 1 at roughly the same time, before either has performed any write operations. Based on this read, both transactions proceed to create a booking and decrement the available seats counter. The result is that two bookings succeed and availableSeats becomes negative (likely -1), causing an overbooking situation.

3.4 Scenario D

ACID Property: Consistency

Explanation:

This scenario demonstrates the **Consistency** property of ACID being properly maintained. Consistency ensures that a transaction brings the database from one valid state to another valid state, maintaining all defined rules, constraints, integrity constraints, and business logic.

In Scenario D, the AI agent attempts to book a trip when availableSeats = 0. Since there are no seats available, allowing this booking would violate a business rule that seats cannot be booked when none are available. The system automatically rejects the transaction, preventing the database from entering an invalid state.

4 Question 4

4.1 Schedule A

Schedule A is a **serial schedule** where T1 executes completely before T2 begins. T1 reads the available seats, decrements the value, writes it back, and inserts the booking. Only after T1 completes does T2 start and perform the same sequence of operations on the updated value.

Since Schedule A is serial, it is both **serializable** and **conflict serializable**. There are no conflicts or race conditions because the transactions do not interleave. T2 reads the correct updated value after T1's write, ensuring both bookings are processed correctly with accurate seat counts. This schedule guarantees correctness but offers no concurrency.

4.2 Schedule B

Schedule B is an interleaved schedule where both T1 and T2 read Trip.availableSeats before either transaction performs any write operations. Both transactions then independently calculate their decremented values based on the same initial read. After both reads are complete, T1 writes its updated value, followed by T2 writing its updated value, and finally both insert their bookings.

This schedule is **NOT conflict serializable** and produces incorrect results. The problem is a classic **lost update**: T2's write overwrites T1's write, causing T1's decrement to be lost.

4.3 Schedule C

Schedule C demonstrates a **dirty read** problem. T2 reads "the updated value from T1" before T1 has actually written it to the database. This means T2 is reading uncommitted, in-memory changes from T1's transaction.

This schedule is **NOT conflict serializable** because T2 is reading data that T1 has not yet committed. This violates isolation. If T1 were to abort after T2's read, T2 would be operating on invalid data that never existed in the database.

5 Question 5

Answer: C. {(11), (12)}

To understand why option C is not possible under ACID guarantees, let's analyze each option:

Option A: {(10), (12)} - T2 executes completely on both tuples: (5)→(10) and (6)→(12), while T1 never runs. This respects atomicity because T2 completed fully (all-or-nothing). Under ACID, it's acceptable for a transaction to not execute at all; what matters is that any transaction that does run must complete entirely. **This is possible.**

Option B: {(11), (13)} - This is the result of serial execution T2→T1: T2 transforms (5)→(10) and (6)→(12), then T1 transforms (10)→(11) and (12)→(13). Both transactions execute completely on all tuples. **This is possible.**

Option C: {(11), (12)} - To achieve (11), we need (5)→T2→(10)→T1→(11). To achieve (12), we need (6)→T2→(12) with no T1 applied. This means T1 was **partially executed**: it updated the first tuple but not the second tuple. This violates the **atomicity** property, which requires that a transaction must be executed completely on all affected data or not at all. T1 cannot selectively update only some tuples. **This is NOT possible.**

Option D: {(12), (14)} - This is the result of serial execution T1→T2: T1 transforms (5)→(6) and (6)→(7), then T2 transforms (6)→(12) and (7)→(14). Both transactions execute completely on all tuples. **This is possible.**