1 (10 points) Short answer

(2 points each) Answer the questions below in 1-2 sentences each.

1. What is a minimum cover of a set of functional dependencies?

> A minimum cover is the smallest set of functional dependencies whose closure contains all the original functional dependencies. In other words: the fewest functional dependencies that can derive the original set of functional dependencies.

2. A table has a B+Tree index on attributes (a, b, c, d). A query specifies `WHERE a=5 AND b<500 AND d>100`. Can this query use the index? Why or why not?

> Yes, although it ignores the `d>100` part. The query can look up the first tuple matching `a=5 AND b=500`, then scan backwards until a=4. Grading: +1 for yes; +1 for describing that it just uses a and b.

3. Describe one kind of query where a B+Tree clearly better than a hash index? Why is it better?

> Range queries. Since B+Trees sort the search keys, range queries over the indexed key only need to scan the matching tuples. With a hash index, you need to scan the entire table. Grading: +1 for range queries; +1 for describing that it only scans matching tuples, or that the hash index must scan all tuples.

4. What is required in order to be able to use an index nested loops join to execute a join between two tables?

> One of the tables must have an index on an attribute being joined. Grading: +1 for "index", +1 for mentioning that it must be on the attribute being joined. Extra: It also must be a "compatible" index type. For an equi-join, both hash and B+Tree indexes work. However, if the join specifies some other predicate (e.g. `WHERE a.age < b.age`), it may only work with a B+Tree.

5. A student who has not taken W4111 creates a program that issues 10 read-only queries. Since their program only reads values, they decide they don't need to use a transaction. Describe one anomaly that could happen because they aren't using a transaction.

> One of:
>
> Unrepeatable reads: If their query reads the same value multiple times, they could see different values.
>
> Non serializable execution: Their transaction could read one value before another transaction modifies it, and another value after a transaction modifies it, causing its results to be inconsistent. For example, with a balance transfer, a read-only transaction could see the "before" state for one account, but the "after" state for another, causing money to appear or disappear.
>
> Grading: +1 for mentioning the right word; +1 for explaining that other write operations can cause surprising behaviour.
>
> Crashes are not an anomaly here: since this transaction doesn't make any updates, it doesn't matter if it crashes in the middle!
>
> Dirty reads are also not an issue: If the updater executes inside a transaction, these 10 read queries won't see uncommitted information.

2 (20 points) Intensive Care Unit Normalization

Suppose you have the following schema representing the duties of nurses in the ICU:

`icu_duty(duty_id, nurse_id, icu_stay_id, therapy_id, patient_id, length, report)`

We denote this schema DNITPLR, representing each attribute by its first letter. Suppose you have the following functional dependencies:

D → NITPLR	`duty_id` is the key for the table
I → PL	Each `icu_stay` applies to one patient and has a fixed duration
N → TP	A nurse always applies the same therapy to the same patient

1. **(4 points)**: For the following example table, are there any rows which violate the stated functional dependencies? If not, explain why. If so, identify the duty_ids, attributes and functional dependency that is violated.

duty_id	**nurse_id**	**icu_stay_id**	**therapy_id**	**patient_id**	**length**	**report**
3000	Evan	5	pain killer	2007	10	report1
3001	Eugene	17	blood pressure	2007	5	report2
3002	Neha	4	blood sample	2003	7	report3
3003	Jinyang	17	blood sample	2007	4	report4
3004	Jinyang	6	temperature	2001	7	report5
3005	Neha	4	blood sample	2003	7	report6

- `duty_ids` 3001 and 3003 contradict the functional dependency I → PL (2 points): `icu_stay_id` 17 has two different stay lengths: 5 and 4. (2 points)
- `duty_ids` 3003 and 3004 contradict the functional dependency N → TP (2 points): Since the value for `nurse_id` is the same, the values for `therapy_id` and `patient_id` must be the same.

Note: `duty_ids` 3000 and 3001 have the same `patient_id`, but different `icu_stays`, so it is acceptable that they have different lengths. Similarly, `duty_ids` 3002 and 3003 have the same `therapy_ids` but different `nurse_ids`. That is also permitted.

2. **(6 points)**: Write a **BCNF** decomposition of this table in the space below. You can denote each table by the first letter of the attributes in it.

From I → PL, we decompose DNITPLR into DNITR and IPL.

From N → TP, we have N → T, we further decompose DNITR into DNIR and NT. We end up with three tables: DNIR, NT, and IPL.

Alternative:

From N → TP, we decompose DNITPLR into DNILR and NTP.

From I → PL, we have I → L, so we can further decompose DNILR into DNIR and IL. We end up with three tables: DNIR, NTP, and IL.

Grading: +4 if they do a single decomposition step; +2 if they get the correct final solution.

3. **(6 points)**: Is the schema you came up with in the previous problem dependency preserving? If so, explain why. If not, why not, and what could be done to address the issue?

For DNIR, NT, IPL: It is not dependency preserving in that you need joins to check N $\rightarrow$ P. Solution: Use 3NF by adding P to NT to get NTP (or add a separate NP table). This preserves dependencies, but with redundancy!

For DNIR, IL, NTP: It is not dependency preserving in that you need joins to check I $\rightarrow$ P. Solution: Use 3NF by adding P to IL to get IPL (or add a separate IP table). This preserves dependencies, but with redundancy!

Grading: +2 for not dependency preserving, +1 for identifying the missing functional dependency, +1 for saying to use 3NF, +2 for the correct 3NF tables.

4. **(4 points)**: Consider a simplified version of this schema with only the attributes DTP, and the projection of the previous functional dependencies: D $\rightarrow$ TP. A consultant decides that it should be decomposed into two tables: DT and TP. Being a good W4111 student, you know that this decomposition does not have the lossless join property.

   Create example tuples for the DTP table to prove to the consultant that the lossless join property does not hold. Write one tuple that is incorrectly produced by the join of DT and TP, that does not exist in your original DTP tuples to prove that this does not hold.

   Hint: Two rows in the DTP table is sufficient to prove this property, but you can use more than that if it makes it easier. It also might be helpful to write out the decomposed DT and TP tables from your example DTP table, but no marks will be given for it.

Grading: +3 for listing two tuples that have the same therapy_id but different patients. +1 for listing a tuple with duty_id that does not match the patient_id.

Example values for DTP:

1	therapy 1	patient x
2	therapy 1	patient y

Decomposed DT:

1	therapy 1
2	therapy 1

Decomposed TP:

therapy 1	patient x
therapy 1	patient y

To join these back together, we need to join on T. The results will be:

1	therapy 1	patient x
1	therapy 1	patient y (wrong)
2	therapy 1	patient x (wrong)
2	therapy 1	patient y

3 (14 points) Transactions and recovery

A W4111 student has created a database to store the number of items sold in a charity auction. The initial sold items table contains the following values:

item_id	price	number
1	10.00	1
2	5.00	0

The charity auction was a success, and the student needs to update the information. She writes two transactions, Transaction A and Transaction B:

Transaction A

```
BEGIN
UPDATE items SET number =
  number + 10 WHERE item_id = 1;
X = SELECT SUM(number) FROM items;
COMMIT;
PRINT X;
```

Transaction B

```
BEGIN
UPDATE items SET number =
  number + 1 WHERE item_id = 2;
X = SELECT SUM(number) FROM items;
COMMIT;
PRINT X;
```

The statement `X = SELECT ...` means the application variable `X` takes the value of the `SELECT` statement on the right. The statement `PRINT X` displays the variable `X` on the screen.

1. **(4 points)**: If the student executes these transactions using a database that correctly implements strict two-phase locking, what are the different possible results printed on the screen for transactions A and B if she executes them at the exact same time?

> Strict two-phase locking means that the database provides serializable consistency. Therefore, the possible results are the results of serial order TA, TB or TB, TA. Extra: It is also possible that this produces a deadlock, if the database that uses strict two-phase locking.
>
> TA, TB: TA prints 11, TB prints 12
>
> TB, TA: TB prints 2, TA prints 12
>
> Grading: 2 points for each correct output.

2. **(4 points)**: The student executes these transactions on a new database called FooDB. It breaks the transactions into individual read and write operations, and executes them in the following interleaved order:

Operation	**Transaction A**	**Transaction B**
1	Read(item_id=1)	
2		Read(item_id=2)
3	Write(item_id=1)	
4		Write(item_id=2)
5	Read(item_id=1)	
6		Read(item_id=1)
7	Read(item_id=2)	
8		Read(item_id=2)
8	Commit	
9		Commit

Is this a serializable order? If yes, what is the equivalent serial order. If not, describe which operation(s) violate serializability.

This is not serializable. Operation 3 and operation 6 implies TA happends before TB. However, operation 4 and operation 7 implies that TB happens before TA. This is a contradiction which means it cannot be serializable.

Grading: 2 points for not serializable. 1 point for each pair of operations (3, 6); (4, 7).

3. **(2 points)**: What would be the result of executing this set of interleaved operations on a database that implements strict two-phase locking with shared read locks and exclusive write locks?

This results in a deadlock at operations 6 and 7. At operation 6, Transaction A holds an exclusive lock on item 1, so Transaction B will wait to acquire a shared lock on it. At operation 7, Transaction B holds an exclusive lock on item 2, so Transaction A will wait to acquire a shared lock on it. At least one of the transactions will need to be aborted.

Grading: 2 points for saying deadlock.

4. **(4 points)** The charity auction uses a database that provides durable transactions, correctly implemented with a write-ahead redo-only log as described in class. The database crashes in the middle of processing some operations. The on-disk state and the redo log are listed below. Reconstruct the state of the database after it recovers using the redo log in the space below.

On Disk State

item_id	number
w	1
x	2
y	3
z	4

Redo Log

transaction	operation
A	w=50
B	y=60
C	x=70
D	z=80
B	COMMIT
C	y=90
A	w=100
C	COMMIT

After Recovery State

item_id	number
w	**1**
x	**70**
y	**90**
z	**4**

In the log, only transactions B and C commit. Therefore, we need to replay the following writes:

y=60, x=70, y=90

Grading: 1 point for each correct value.

4 (26 points) Query execution

To store a database of movies and actors, we create the following tables:

```
CREATE TABLE movies(
  m_id INTEGER PRIMARY KEY,
  m_name TEXT NOT NULL
);

CREATE TABLE actors(
  a_id INTEGER PRIMARY KEY,
  a_name TEXT NOT NULL
);

CREATE TABLE acts_in(
  m_id INTEGER REFERENCES movies
  a_id INTEGER REFERENCES actors
  PRIMARY KEY(m_id, a_id)
);
```

The tables are all stored in unordered heap files, without any indexes. For these questions, we will consider the cost to read a page from disk as being P, and assume the database must always read tables from disk. We will ignore all other costs.

1. **(2 points)**: What will the average cost be to execute the following query?

   ```
   SELECT * FROM actors WHERE a_id = 1042;
   ```

   Since `actors.a_id` is a primary key, there is a single unique value. Assuming the id is in the table, on average we need to scan half the rows, so the cost is:

   $$\frac{\text{pages}}{2} \times P$$

   Note: If the id is not on the table, we do need to scan all rows, but that is a less common case. A query optimizer would likely choose to estimate half the pages instead.

2. **(2 points)**: What will the average cost be to execute the following query?

   ```
   SELECT * FROM actors WHERE 1042 <= a_id and a_id <= 5072;
   ```

   We must scan all the rows to find all the matching records so the cost is pages $\times P$.

Hint: the following three parts are related; read all three before answering

3. **(4 points)**: We create a primary B+Tree index on `actors(a_id)`. Assume the height of the tree is 2 (2 levels of directory pages then the leaf pages). What is the average cost for the query in the previous question? Assume the query matches 1% of the tuples in the actors table, and the leaf nodes in the primary index occupy 10000 pages.

This question was intended to refer only to the query from Q4.2 (the immediately previous question).

Cost to read the directory pages (which includes the root): 2
Cost to read matching leaf pages: $0.01 \times 10000 - 1 = 99$ (since it matches 1% of the tuples, we can assume it matches approximately 1% of the pages)
Total: $102 \times P$ ($103P$ is also acceptable, due to an an off-by-one error)

4. **(4 points)**: What would the cost of the query in the previous two questions be if we create a *secondary* B+Tree index on `actors(a_id)`? Assume the query matches 1% of the tuples in the actors table, and 10 leaf pages in the secondary index, and the unordered heap file occupies 8000 pages.

This question was intended to refer only to the query from Q4.2 (since Q4.3 references the same query). It also should have specified that the height of the secondary index is the same: 2 levels (likely true). To produce a *correct* estimate, we also need to know the number of tuples per leaf page. If the 10 leaf pages contain 1000 tuples, we will probably read 1000 pages in the heap file.

However, due to this error in the question, it is reasonable to assume that since the range matches 1% of the tuples, we will on average need to read 1% of the unordered heap file leaf pages. This is not correct. It is in fact very likely that 1% of the tuples is way more than 80 tuples, so it can easily read a large number of leaf pages!

Cost to read the directory pages: 2
Cost to read the leaf pages: 10
Cost to read the actual tuples: $0.01 \times$ actor tuples
Correct total: $(12 + 0.01 \times \text{actor tuples}) \times P$
Acceptable total: $(2 + 10 + 0.01 \times 8000) \times P = 92 \times P$ ($93 \times P$ is also acceptable due to off-by-one errors)

Grading: -1 for saying 1 leaf page instead of 10.

5. **(4 points)**: According to this simple model, the cost for answering this range query with a secondary index or a primary index are similar. This will not likely be true in reality. Which is probably faster for this query in a real system? Why?

In reality, the primary index would be much better for a range scan because the I/O is mostly sequential, while the I/O for a secondary index is mostly random. Sequential I/O is much more efficient on real disks or in memory.

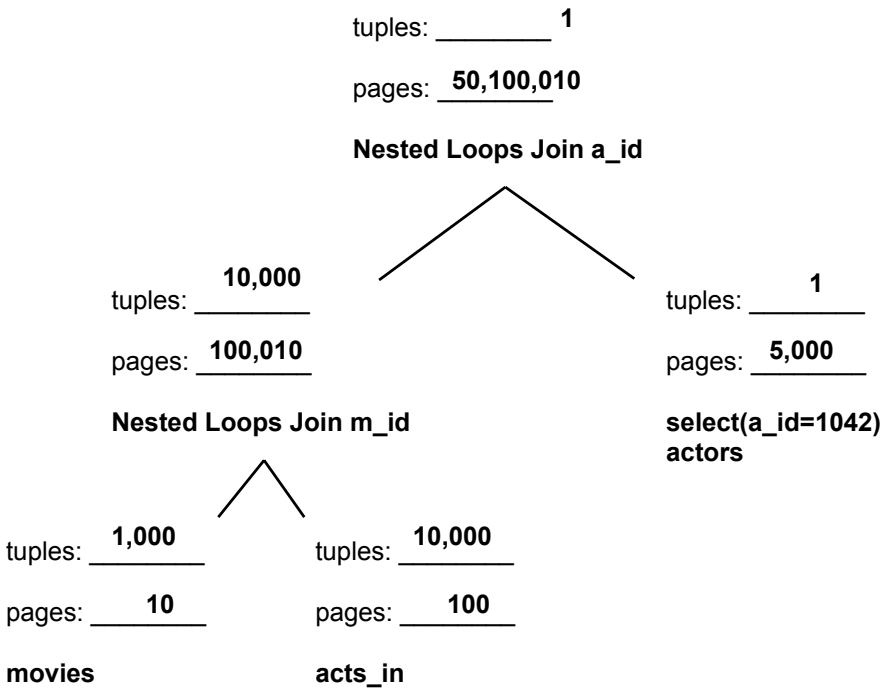
The key part of this question was that it was asking about *this specific query*. We gave part marks for saying that a primary index is better because for a large range, the secondary index will read far more pages.

6. **(10 points)**: Assume that we have no indexes, and our only join algorithm is a nested loops join. We execute the following query:

```
SELECT m_name, a_name
FROM movies, actors, acts_in
WHERE movies.m_id = acts_in.m_id AND actors.a_id = acts_in.a_id
  AND actors.a_id = 1042;
```

The database optimizer decides to execute the query with the following query plan. For each table and operator, estimate the number of output tuples and the total number of pages read up to that point in the query plan. (Note: this means the top pages value is the total for the entire query.) Assume that id values are uniformly distributed. The tables have the following number of rows and pages on disk:

Table	Rows	Pages
movies	1,000 rows	10 pages
actors	100,000 rows	10,000 pages
acts_in	10,000 rows	100 pages



Write any assumptions or calculations you make below:

Grading: 1 point for each valid answer.

For movies and acts_in, the entire tables must be scanned, so those operators produce all tuples and pages. For movies: 1000 tuples and 10 pages. For acts_in: 10,000 tuples and 100 pages.

For movies join acts_in: The acts_in table must be scanned once for every tuple in movies, which costs 1000 tuples in outer $\times$100 pages in acts_in. You also need to add the cost for reading movies once, for a total of 100,010 pages. How many tuples does this produce? Well, in the schema: acts_in has a foreign key constraint on movies.m_id. This means every acts_in.m_id matches exactly 1 value in movies. Therefore, this produces 10,000 tuples: the size of acts_in.

For select on actors: We expected you to use your answer from Q4.1. Ideally: 1 tuple and 5,000 pages.

For the final join: This costs 10,000 tuples $\times$ the cost of the actors table, so $10,000 \times 5,000$ plus the cost of the outer (which is the previous answer). Therefore, the correct total is $10000 \times 5000 + 100010 = 50,100,010$. It is also acceptable to assume the single result from the inner scan can be cached, so the cost is then $5,000 + 100,010 = 105,010$.

For the final number of tuples: there are a number of possible estimates. Due to the foreign key constraint on acts_in, we know that there must be at least 1 matching actor for each acts_in, which means that we have a maximum possible output size of 10,000. However, it is also possible that this specific actor matches 0 tuples. Since there are 10^5 actors and 10^4 acts in, it seems reasonable to estimate that each actor tuple matches $10^4/10^5 = 0.1$ acts_in tuples, which is probably the "best" estimate, which either rounds to 0 or 1. However, we accepted any "reasonable" estimate anywhere within the range 10,000 to 0.