CISC 7510X Midterm Exam

Name: _____ Date: _____

For the below questions, use the following Pizzeria schema definition.

```
customer (customer_id, customer_name, city, signup_date)
pizza (pizza_id, pizza_name, category, price)
orders (order_id, customer_id, order_date, order_type, tip)
order_item (order_id, pizza_id, quantity)
```

Each order has a customer. Each `order_item` row records how many pizzas of one kind were on one order; the amount for the line is `quantity` × `price`. Pizza category is meat, veggie, etc. order type is delivery vs pickup, etc. tip is null for pickup orders. Pick the best answer that fits the question. Not all of the answers may be correct. If none of the answers fit, write your own answer (answer E). There are at *most* 2 questions where writing your own answer may be appropriate.

1. Show the name and price of every pizza in the `meat` or `specialty` category that costs less than $15.00.

(a)
```
SELECT pizza_name, price FROM pizza
WHERE category = 'meat' OR category = 'specialty' AND price < 15
```

(b)
```
SELECT pizza_name, price FROM pizza
WHERE category IN ('meat', 'specialty') AND price < 15
```

(c)
```
SELECT pizza_name, price FROM pizza
WHERE category IN ('meat', 'specialty') OR price < 15
```

(d)
```
SELECT category, price FROM pizza
WHERE category IN ('meat', 'specialty') AND price < 15
```

(e)

2. List the order id and order date of every order placed in February 2026, with the most recent order first.

(a)
```
SELECT order_id, order_date FROM orders
WHERE order_date BETWEEN '2026-02-01' AND '2026-03-01'
ORDER BY order_date DESC;
```

(b)
```
SELECT order_id, order_date FROM orders
WHERE order_date >= '2026-02-01' AND order_date < '2026-03-01'
ORDER BY order_date;
```

(c)
```
SELECT order_id, order_date FROM orders
WHERE order_date >= '2026-02-01' OR order_date < '2026-03-01'
ORDER BY order_date DESC;
```

(d)
```
SELECT order_id, order_date FROM orders
WHERE order_date >= '2026-02-01' AND order_date < '2026-03-01'
ORDER BY order_date DESC;
```

(e)

3. How many orders have a recorded tip (that is, a tip that is not `NULL`)?

(a) `SELECT COUNT(*) FROM orders WHERE tip IS NOT NULL;`
(b) `SELECT COUNT(*) FROM orders WHERE tip <> NULL;`
(c) `SELECT COUNT(DISTINCT tip) FROM orders;`
(d) `SELECT COUNT(*) FROM orders;`
(e)

4. List the name of every customer who has placed at least one order, showing each name only once.

(a)
```
SELECT c.customer_name
FROM customer c
  JOIN orders o ON c.customer_id = o.customer_id;
```
(b)
```
SELECT DISTINCT c.customer_name
FROM customer c
  LEFT JOIN orders o ON c.customer_id = o.customer_id;
```
(c)
```
SELECT DISTINCT c.customer_name
FROM customer c
  JOIN orders o ON c.customer_id = o.customer_id;
```
(d)
```
SELECT DISTINCT c.customer_name
FROM customer c
  JOIN orders o ON c.customer_id = o.order_id;
```
(e)

5. For each pizza category, show the category and the total number of pizzas sold (the sum of `quantity`) across all orders.

(a)
```
SELECT p.category, COUNT(*)
FROM pizza p
  JOIN order_item oi ON p.pizza_id = oi.pizza_id
GROUP BY p.category;
```
(b)
```
SELECT p.category, SUM(oi.quantity)
FROM pizza p
  JOIN order_item oi ON p.pizza_id = oi.pizza_id
GROUP BY p.category;
```
(c)
```
SELECT p.pizza_name, SUM(oi.quantity)
FROM pizza p
  JOIN order_item oi ON p.pizza_id = oi.pizza_id
GROUP BY p.pizza_name;
```
(d)
```
SELECT p.category, SUM(p.price)
FROM pizza p
  JOIN order_item oi ON p.pizza_id = oi.pizza_id
GROUP BY p.category;
```
(e)

6. For every customer whose total spending exceeds \$90, show the customer's name and the total spent. Spending is the sum of `quantity` × `price` over all of the customer's order items.

(a)
```
SELECT c.customer_name, SUM(p.price) AS total_spent
FROM customer c
JOIN orders o ON c.customer_id = o.customer_id
JOIN order_item oi ON o.order_id = oi.order_id
JOIN pizza p ON oi.pizza_id = p.pizza_id
GROUP BY c.customer_name
HAVING SUM(p.price) > 90;
```

(b)
```
SELECT c.customer_name, SUM(oi.quantity * p.price) AS total_spent
FROM customer c
JOIN orders o ON c.customer_id = o.customer_id
JOIN order_item oi ON o.order_id = oi.order_id
JOIN pizza p ON oi.pizza_id = p.pizza_id
WHERE oi.quantity * p.price > 90
GROUP BY c.customer_name;
```

(c)
```
SELECT c.customer_name, SUM(oi.quantity * p.price) AS total_spent
FROM customer c
JOIN orders o ON c.customer_id = o.customer_id
JOIN order_item oi ON o.order_id = oi.order_id
JOIN pizza p ON oi.pizza_id = p.pizza_id
GROUP BY c.customer_name
HAVING SUM(oi.quantity * p.price) > 90;
```

(d)
```
SELECT c.customer_name, SUM(oi.quantity * p.price) AS total_spent
FROM customer c
JOIN orders o ON c.customer_id = o.customer_id
JOIN order_item oi ON o.order_id = oi.order_id
JOIN pizza p ON p.pizza_id = o.order_id
GROUP BY c.customer_name
HAVING SUM(oi.quantity * p.price) > 90;
```

(e)

7. Show every customer's name together with the number of orders they have placed. Customers who have never ordered must still appear, with a count of 0.

(a)
```
SELECT c.customer_name, COUNT(o.order_id) AS num_orders
FROM customer c
LEFT JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_name;
```

(b)
```
SELECT c.customer_name, COUNT(*) AS num_orders
FROM customer c
LEFT JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_name;
```

(c)
```
SELECT c.customer_name, COUNT(o.order_id) AS num_orders
FROM customer c
JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_name;
```

(d)
```
SELECT c.customer_name, COUNT(o.order_id) AS num_orders
```

```
    FROM customer c
    LEFT JOIN orders o ON c.customer_id = o.customer_id
                      AND o.order_type = 'delivery'
    GROUP BY c.customer_name;
(e)
```

8. For each customer city, show the city, the number of delivery orders, and the number of pickup orders as two separate columns.

```
(a) SELECT c.city,
      COUNT(CASE WHEN o.order_type = 'delivery' THEN 1 ELSE 0 END) AS deliveries
    ,
      COUNT(CASE WHEN o.order_type = 'pickup' THEN 1 ELSE 0 END) AS pickups
    FROM customer c
    JOIN orders o ON c.customer_id = o.customer_id
    GROUP BY c.city;
(b) SELECT c.city, o.order_type, COUNT(*) AS num_orders
    FROM customer c
    JOIN orders o ON c.customer_id = o.customer_id
    GROUP BY c.city, o.order_type;
(c) SELECT c.city,
      SUM(CASE WHEN o.order_type = 'delivery' THEN 1 ELSE 0 END) AS deliveries,
      SUM(CASE WHEN o.order_type <> 'pickup' THEN 1 ELSE 0 END) AS pickups
    FROM customer c
    JOIN orders o ON c.customer_id = o.customer_id
    GROUP BY c.city;
(d) SELECT c.city,
      SUM(CASE WHEN o.order_type = 'delivery' THEN 1 ELSE 0 END) AS deliveries,
      SUM(CASE WHEN o.order_type = 'pickup' THEN 1 ELSE 0 END) AS pickups
    FROM customer c
    JOIN orders o ON c.customer_id = o.customer_id
    GROUP BY c.city;
(e)
```

9. Show the name and total spending of each customer whose total spending is greater than the average total spending of customers who have placed orders.

```
(a) WITH spend AS (
      SELECT o.customer_id, SUM(oi.quantity * p.price) AS total
      FROM orders o
      JOIN order_item oi ON o.order_id = oi.order_id
      JOIN pizza p ON oi.pizza_id = p.pizza_id
      GROUP BY o.customer_id
    )
    SELECT c.customer_name, s.total
    FROM customer c
    JOIN spend s ON c.customer_id = s.customer_id
    WHERE s.total > (SELECT AVG(price) FROM pizza);

(b) WITH spend AS (
```

```
    SELECT o.customer_id, SUM(oi.quantity * p.price) AS total
    FROM orders o
    JOIN order_item oi ON o.order_id = oi.order_id
    JOIN pizza p ON oi.pizza_id = p.pizza_id
    GROUP BY o.customer_id
  )
  SELECT c.customer_name, s.total
  FROM customer c
  JOIN spend s ON c.customer_id = s.customer_id
  WHERE s.total > (SELECT AVG(total) FROM spend);
```

(c)
```
WITH spend AS (
    SELECT o.customer_id, SUM(oi.quantity * p.price) AS total
    FROM orders o
    JOIN order_item oi ON o.order_id = oi.order_id
    JOIN pizza p ON oi.pizza_id = p.pizza_id
    GROUP BY o.customer_id
  )
  SELECT c.customer_name, s.total
  FROM customer c
  JOIN spend s ON c.customer_id = s.customer_id
  WHERE s.total > (SELECT SUM(total) / (SELECT COUNT(*) FROM customer)
                   FROM spend);
```

(d)
```
WITH spend AS (
    SELECT o.customer_id, SUM(oi.quantity * p.price) AS total
    FROM orders o
    JOIN order_item oi ON o.order_id = oi.order_id
    JOIN pizza p ON oi.pizza_id = p.pizza_id
    GROUP BY o.customer_id
  )
  SELECT c.customer_name, s.total
  FROM customer c
  JOIN spend s ON c.customer_id = s.customer_id
  WHERE s.total < (SELECT AVG(total) FROM spend);
```

(e)

10. List the name of every customer who has never ordered a pizza from the `meat` category (including customers who have never placed any order).

(a)
```
SELECT DISTINCT c.customer_name
FROM customer c
JOIN orders o ON c.customer_id = o.customer_id
JOIN order_item oi ON o.order_id = oi.order_id
JOIN pizza p ON oi.pizza_id = p.pizza_id
WHERE p.category <> 'meat';
```

(b)
```
SELECT c.customer_name
FROM customer c
WHERE c.customer_id NOT IN (
```

```
      SELECT o.customer_id
      FROM orders o
      JOIN order_item oi ON o.order_id = oi.order_id
      JOIN pizza p ON oi.pizza_id = p.pizza_id
      WHERE p.category <> 'meat'
    );
```

(c)
```
SELECT c.customer_name
FROM customer c
WHERE NOT EXISTS (
  SELECT 1
  FROM orders o
  JOIN order_item oi ON o.order_id = oi.order_id
  JOIN pizza p ON oi.pizza_id = p.pizza_id
  WHERE o.customer_id = c.customer_id AND p.category = 'meat'
);
```

(d)
```
SELECT DISTINCT c.customer_name
FROM customer c
LEFT JOIN orders o ON c.customer_id = o.customer_id
LEFT JOIN order_item oi ON o.order_id = oi.order_id
LEFT JOIN pizza p ON oi.pizza_id = p.pizza_id
WHERE p.category <> 'meat' OR p.category IS NULL;
```

(e)

11. For each customer, show the customer id, order id, and order date of that customer's most recent order.

(a)
```
SELECT customer_id, order_id, order_date
FROM (SELECT customer_id, order_id, order_date,
             ROW_NUMBER() OVER (PARTITION BY customer_id
                                ORDER BY order_date DESC) AS rn
      FROM orders) t
WHERE rn = 1;
```

(b)
```
SELECT customer_id, order_id, order_date
FROM (SELECT customer_id, order_id, order_date,
             ROW_NUMBER() OVER (PARTITION BY customer_id
                                ORDER BY order_date) AS rn
      FROM orders) t
WHERE rn = 1;
```

(c)
```
SELECT customer_id, order_id, order_date
FROM (SELECT customer_id, order_id, order_date,
             ROW_NUMBER() OVER (ORDER BY order_date DESC) AS rn
      FROM orders) t
WHERE rn = 1;
```

(d)
```
SELECT customer_id, order_id, order_date
FROM (SELECT customer_id, order_id, order_date,
             ROW_NUMBER() OVER (PARTITION BY order_date
```

```
                                    ORDER BY customer_id) AS rn
        FROM orders) t
    WHERE rn = 1;
```

(e)

12. For every order, show the order id, customer id, order date, and the date of that same customer's previous order (NULL if it is the customer's first order).

```
(a) SELECT order_id, customer_id, order_date,
           LEAD(order_date) OVER (PARTITION BY customer_id
                                  ORDER BY order_date) AS prev_date
    FROM orders;
```

```
(b) SELECT order_id, customer_id, order_date,
           LAG(order_date) OVER (ORDER BY order_date) AS prev_date
    FROM orders;
```

```
(c) SELECT order_id, customer_id, order_date,
           LAG(order_date, 2) OVER (PARTITION BY customer_id
                                    ORDER BY order_date) AS prev_date
    FROM orders;
```

```
(d) SELECT order_id, customer_id, order_date,
           LAG(order_date) OVER (PARTITION BY customer_id
                                 ORDER BY order_date) AS prev_date
    FROM orders;
```

(e)

13. For every order, show the order id, order date, and the date of that customer's most recent order overall (so every row for the same customer shows the same value).

```
(a) SELECT order_id, order_date,
           LAST_VALUE(order_date) OVER (PARTITION BY customer_id
                                        ORDER BY order_date) AS latest
    FROM orders;
```

```
(b) SELECT order_id, order_date,
           FIRST_VALUE(order_date) OVER (PARTITION BY customer_id
                                         ORDER BY order_date
                                         ROWS BETWEEN UNBOUNDED PRECEDING
                                                  AND UNBOUNDED FOLLOWING) AS
    latest
    FROM orders;
```

```
(c) SELECT order_id, order_date,
           LAST_VALUE(order_date) OVER (PARTITION BY customer_id
                                        ORDER BY order_date
                                        ROWS BETWEEN UNBOUNDED PRECEDING
                                                 AND UNBOUNDED FOLLOWING) AS
    latest
    FROM orders;
```

(d)
```
SELECT order_id, order_date,
       LAST_VALUE(order_date) OVER (ORDER BY order_date
                                    ROWS BETWEEN UNBOUNDED PRECEDING
                                             AND UNBOUNDED FOLLOWING) AS
    latest
    FROM orders;
```

(e)

14. For each customer who has placed orders, show the customer id, their total spending, and the percentage of all customers' combined spending that this represents, rounded to one decimal place.

(a)
```
SELECT o.customer_id,
       SUM(oi.quantity * p.price) AS spend,
       ROUND(100.0 * SUM(oi.quantity * p.price)
             / SUM(SUM(oi.quantity * p.price))
                 OVER (PARTITION BY o.customer_id), 1) AS pct
    FROM orders o
    JOIN order_item oi ON o.order_id = oi.order_id
    JOIN pizza p ON oi.pizza_id = p.pizza_id
    GROUP BY o.customer_id;
```

(b)
```
SELECT o.customer_id,
       SUM(oi.quantity * p.price) AS spend,
       ROUND(100.0 * SUM(oi.quantity * p.price)
             / SUM(SUM(oi.quantity * p.price)) OVER (), 1) AS pct
    FROM orders o
    JOIN order_item oi ON o.order_id = oi.order_id
    JOIN pizza p ON oi.pizza_id = p.pizza_id
    GROUP BY o.customer_id;
```

(c)
```
SELECT o.customer_id,
       SUM(oi.quantity * p.price) AS spend,
       ROUND(100.0 * SUM(oi.quantity * p.price)
             / SUM(SUM(oi.quantity * p.price))
                 OVER (ORDER BY o.customer_id), 1) AS pct
    FROM orders o
    JOIN order_item oi ON o.order_id = oi.order_id
    JOIN pizza p ON oi.pizza_id = p.pizza_id
    GROUP BY o.customer_id;
```

(d)
```
SELECT o.customer_id,
       SUM(oi.quantity * p.price) AS spend,
       ROUND(100.0 * SUM(oi.quantity * p.price)
             / AVG(SUM(oi.quantity * p.price)) OVER (), 1) AS pct
    FROM orders o
    JOIN order_item oi ON o.order_id = oi.order_id
    JOIN pizza p ON oi.pizza_id = p.pizza_id
    GROUP BY o.customer_id;
```

(e)

15. For each pizza category, show the best-selling pizza by total quantity sold, including every pizza that ties for first place. Show the category, pizza name, and total quantity.

(a)
```
SELECT category, pizza_name, total_qty
FROM (SELECT p.category, p.pizza_name,
             SUM(oi.quantity) AS total_qty,
             RANK() OVER (PARTITION BY p.category
                          ORDER BY SUM(oi.quantity) DESC) AS rnk
      FROM pizza p
      JOIN order_item oi ON p.pizza_id = oi.pizza_id
      GROUP BY p.category, p.pizza_name) t
WHERE rnk = 1;
```

(b)
```
SELECT category, pizza_name, total_qty
FROM (SELECT p.category, p.pizza_name,
             SUM(oi.quantity) AS total_qty,
             ROW_NUMBER() OVER (PARTITION BY p.category
                                ORDER BY SUM(oi.quantity) DESC) AS rnk
      FROM pizza p
      JOIN order_item oi ON p.pizza_id = oi.pizza_id
      GROUP BY p.category, p.pizza_name) t
WHERE rnk = 1;
```

(c)
```
SELECT category, pizza_name, total_qty
FROM (SELECT p.category, p.pizza_name,
             SUM(oi.quantity) AS total_qty,
             RANK() OVER (ORDER BY SUM(oi.quantity) DESC) AS rnk
      FROM pizza p
      JOIN order_item oi ON p.pizza_id = oi.pizza_id
      GROUP BY p.category, p.pizza_name) t
WHERE rnk = 1;
```

(d)
```
SELECT category, pizza_name, total_qty
FROM (SELECT p.category, p.pizza_name,
             SUM(oi.quantity) AS total_qty,
             RANK() OVER (PARTITION BY p.category
                          ORDER BY SUM(oi.quantity)) AS rnk
      FROM pizza p
      JOIN order_item oi ON p.pizza_id = oi.pizza_id
      GROUP BY p.category, p.pizza_name) t
WHERE rnk = 1;
```

(e)

16. Transaction A reads the price of pizza 1, and a few seconds later reads it again. In between, transaction B updates that price and commits, so A sees two different values within the same transaction. What is the *lowest* standard isolation level at which this cannot happen?

(a) READ UNCOMMITTED

(b) READ COMMITTED

(c) REPEATABLE READ

(d) SERIALIZABLE

(e)

17. Under row-level locking, transaction T1 updates pizza 1 and then pizza 2, while T2 concurrently updates pizza 2 and then pizza 1. T1 holds the lock on pizza 1 and T2 holds the lock on pizza 2 when each requests the other's row. What happens, and how can the application avoid it?

(a) Both requests are granted, because update locks are shared. Nothing needs to be done.

(b) A deadlock occurs; the DBMS typically detects the cycle and aborts one transaction. Having every transaction update rows in the same order (e.g., by `pizza_id`) prevents it.

(c) Both transactions wait forever, since no DBMS can detect such a cycle. Only restarting the server resolves it.

(d) T2 silently skips its update of pizza 1, because T1 started earlier. Lowering the isolation level prevents this.

(e)

18. A table `ORDER_LINE(order_id, pizza_id, pizza_name, quantity, customer_id, customer_city)` has primary key `(order_id, pizza_id)` and the functional dependencies `pizza_id` $\rightarrow$ `pizza_name`, `order_id` $\rightarrow$ `customer_id`, and `customer_id` $\rightarrow$ `customer_city`. All column values are atomic. Which statement is correct?

(a) It violates 1NF because the primary key consists of two columns.

(b) It satisfies 2NF but violates 3NF, because `customer_city` depends on the key only transitively.

(c) It satisfies BCNF, because every column holds a single atomic value.

(d) It is in 1NF but not in 2NF, because `pizza_name` depends on only part of the primary key.

(e)

19. A table `CUSTOMER_ADDR(customer_id, zip, city)` has primary key `customer_id` and the functional dependency `zip` $\rightarrow$ `city`. Which decomposition removes the redundancy, guarantees a lossless join, and preserves the dependency `zip` $\rightarrow$ `city`?

(a) R1(`customer_id`, `zip`) and R2(`zip`, `city`)

(b) R1(`customer_id`, `city`) and R2(`zip`, `city`)

(c) R1(`customer_id`, `zip`) and R2(`customer_id`, `city`)

(d) No decomposition is needed, because `customer_id` is the key and the table is therefore already free of redundancy.

(e)

20. The table `orders` has a single B-tree index on `(customer_id, order_date)` and no other indexes. Which query can use this index to jump directly to a small range of index entries, rather than scanning the whole table or the whole index?

(a) `SELECT * FROM orders WHERE tip > 3;`

(b) `SELECT * FROM orders WHERE order_date >= '2026-02-01';`

(c) `SELECT * FROM orders WHERE order_type = 'pickup';`

(d) `SELECT * FROM orders WHERE customer_id = 2 AND order_date >= '2026-02-01';`

(e)