Odne Rindheim
With corrected task 4.1, 6.3 and 6.5

Task 1: Triggers

Logged into mariadb:
Mysql -u odnerindheim -p
Created database:
CREATE DATABASE a4t1;
USE a4t1;
Create table TheTable:
```
CREATE TABLE TheTable (
        Id INT PRIMARY KEY,
        Name VARCHAR(50),
        Age INT
        );
```
Create table LogTable:
```
CREATE TABLE LogTable (
        Log_id INT PRIMARY KEY AUTO_INCREMENT,
        Action VARCHAR(10),
        Table_name VARCHAR(50),
        Old_value TEXT,
        New_value TEXT,
        Change_timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        );
```
Setting Triggers:
```
DELIMITER //

CREATE TRIGGER log_insert
AFTER INSERT ON TheTable
FOR EACH ROW
BEGIN
  INSERT INTO LogTable (action, table_name, new_values)
  VALUES ('INSERT', 'TheTable', CONCAT('id: ', NEW.id, ', name: ', NEW.name, ', age: ',
NEW.age));
END//

CREATE TRIGGER log_update
AFTER UPDATE ON TheTable
FOR EACH ROW
BEGIN
  INSERT INTO LogTable (action, table_name, old_values, new_values)
  VALUES ('UPDATE', 'TheTable', CONCAT('id: ', OLD.id, ', name: ', OLD.name, ', age: ',
OLD.age),
                  CONCAT('id: ', NEW.id, ', name: ', NEW.name, ', age: ', NEW.age));
END//

CREATE TRIGGER log_delete
```

Odne Rindheim
With corrected task 4.1, 6.3 and 6.5

```
AFTER DELETE ON TheTable
FOR EACH ROW
BEGIN
   INSERT INTO LogTable (action, table_name, old_values)
   VALUES ('DELETE', 'TheTable', CONCAT('id: ', OLD.id, ', name: ', OLD.name, ', age: ',
OLD.age));
END//

DELIMITER ;
```

Demonstration:

```
INSERT INTO TheTable (id, name, age) VALUES (1, ‘Odne’, 19);
UPDATE TheTable SET age = 20 WHERE id  1;
DELETE FROM TheTable WHERE id = 1;
SELECT * FROM LogTable;
```

Output:

Log_id	Action	Table_name	Old_value	New_value	Change_timestamp
1	INSERT	TheTable	NULL	Id: 1, name: Odne, age: 19	2024-02-28 18:47:57
2	UPDATE	TheTable	Id: 1, Name: Odne, Age: 19	Id: 1, Name: Odne, Age: 20	2024-02-28 18:48:10
3	DELETE	TheTable	Id: 1, Name: Odne, Age: 20	NULL	2024-02-28 18:48:20

Odne Rindheim
With corrected task 4.1, 6.3 and 6.5

Task 2: Temporal Database

Created database:
CREATE DATABASE a4t2;
USE a4t2;

Creating AnotherTable:
CREATE TABLE AnotherTable (
    Id INT PRIMARY KEY,
    Name VARCHAR(50),
    Age INT,
    Valid_from TIMESTAMP(6) GENERATED ALWAYS AS ROW START,
    Valid_to TIMESTAMP(6) GENERATED ALWAYS AS ROW END,
) WITH SYSTEM VERSIONING;

Inserting data:
INSERT INTO AnotherTable (id, name, age) VALUES
    (1, 'doghaus', 30),
    (2, 'partyhaus', 35),
    (3, 'patriothaus', 25);

Doing modifications:
UPDATE AnotherTable SET age = 32 WHERE id = 1;
DELETE FROM AnotherTable WHERE id = 2;
INSERT INTO AnotherTable (id, name, age) VALUES (4, 'safehaus', 27);

Query with 'FOR SYSTEM_TIME AS OF':
SELECT * FROM AnotherTable FOR SYSTEM_TIME AS OF '2024-02-28 00:00:00;
Output: Empty set.
SELECT * FROM AnotherTable FOR SYSTEM_TIME AS OF '2024-02-28 23:59:59;
Output:

Id	Name	Age	Valid_from	Valid_to
1	Doghaus	32	2024-02-28 19:18:44	2038-01-19 04:14:07
2	Patriothaus	25	2024-02-28 19:18:26	2028-01-19 04:14:07
4	Safehaus	27	2024-02-28 19:19:20	2028-01-19 04:14:07

Odne Rindheim
With corrected task 4.1, 6.3 and 6.5

Task 3: Integrity Constraint

Created database:
CREATE DATABASE a4t3;
USE a4t3;

Creating table teacher:

```
CREATE TABLE teacher (
    Teacher_id INT PRIMARY KEY AUTO_INCREMENT,
    Name VARCHAR(50),
    Salary DECIMAL(10,2) CHECK ( salary BETWEEN 1000 AND 100000),
    Bonus DECIMAL(10,2)
    Total DECIMAL(10,2) GENERATED ALWAYS AS (salary + bonus) stored
    );
```

Insert data:

```
INSERT INTO teacher (name, salary, bonus) values
    ('John', 50000.00, 5000.00),
    ('Alice', 75000.00, 5000.00),
    ('Bob', 30000.00, 2000.00);
```

Query the table:
SELECT * FROM teacher;
Output:

Teacher_id	Name	Salary	Bonus	Total
1	John	50000.00	5000.00	55000.00
2	Alice	75000.00	10000.00	85000.00
3	Bob	30000.00	2000.00	32000.00

The table satisfies the conditions of 3NF, as there are no transitive dependencies present.

Odne Rindheim
With corrected task 4.1, 6.3 and 6.5

Task 4: Order of triggers

1.
Create database:
CREATE DATABASE a4t4;
USE a4t4;

Create tables T1, T2, T3:
CREATE TABLES T1 (id INT UNSIGNED PRIMARY KEY);
CREATE TABLES T2 (id INT UNSIGNED PRIMARY KEY);
CREATE TABLES T3 (id INT UNSIGNED, source_table VARCHAR(2), PRIMARY KEY (id, source_table));

Create trigger tr12, tr23, tr13:
Delimiter //

```
CREATE TRIGGER tr12 BEFORE INSERT ON T1
    FOR EACH ROW
    BEGIN
        INSERT INTO T2 (id) VALUES (NEW.id);
    END//

CREATE TRIGGER tr23 AFTER INSERT ON T2
    FOR EACH ROW
    BEGIN
        INSERT INTO T3 (id, source_table) VALUES (NEW.id, 'T2');
    END//

CREATE TRIGGER tr13 AFTER INSERT ON T1
    FOR EACH ROW
    BEGIN
        INSERT INTO T3 (id, source_table) VALUES (NEW.id, 'T1');
    END//

DELIMITER ;
```

Insert a row into T1:
INSERT INTO T1 (id) VALUES (1);

Upon inserting a row into 'T1', the triggers are executed in the following order:
First, 'tr12' is fired before the insert operation on 'T1' is completed. This trigger inserts a corresponsing row into 'T2'.
Next, upon the completion of the insert into 'T2', 'tr23' is triggered, inserting into 'T3' and marking the insert as coming from 'T2'.

Odne Rindheim
With corrected task 4.1, 6.3 and 6.5

Finally, after the insert into 'T1' is completed, 'tr13' is activated, inserting into 'T3' but marking this insert as coming from 'T1'

The 'source_table' column in 'T3' allows both 'tr23' and 'tr13' to insert into 'T3' successfully, cirumventing the issue of primary key conflixts. This modification demonstrates a clear understanding of how to manage trigger operations and their execution order, ensuring that the database shcema supports the required functionality without errors.

2. Unexpected results or deadlocks can occur if triggers interact with the same data in conflicting ways or if there are circular dependencies between triggers.

Odne Rindheim
With corrected task 4.1, 6.3 and 6.5

Task 5: Pendant DELETE

Create database:
CREATE DATABASE a4t5;
USE a4t5;

Create tables:
CREATE TABLE Parent (id INT PRIMARY KEY);

```
Create Table Child (
	Id INT PRIMARY KEY,
	Parent_id INT,
	FOREIGN KEY (parent_id) REFERENCES Parent(id) ON DELETE CASCADE
	);
```

Inserting rows to each table:

INSERT INTO Parent (id) VALUES (1), (2);
INSERT INTO Child (id, parent_id) VALUES (101, 1), (102,1), (103,2);

Creating Trigger:

```
DELIMITER //

CREATE TRIGGER pendant_delete_trigger
	AFTER DELETE ON Child
	FOR EACH ROW
	BEGIN
	DECLARE child_count INT;
	SELECT COUNT (*) INTO child_count FROM Child WHERE parent_id =
OLD.parent_id;
	IF child_count = 0 THEN
		DELETE FROM Parent WHERE id = OLD.parent_id;
	END IF;
END//

DELIMITER ;
```

Implementing the pendant DELETE rule using triggers requires careful consideration of the trigger logic to ensure correct behavior. Compared to other rules like CASCADE and SET NULL, pendant DELETE involves more complex logic to determine when to the delete the parent row based on the presence of child rows. Managing the deletion of parent rows based on child row counts adds an additional layer of complexity to database design and trigger implementation.

Odne Rindheim
With corrected task 4.1, 6.3 and 6.5

Task 6: Concurrency

Creating database:
CREATE DATABASE a4t6;
USE a4t6;

1. Create Normalized Database Schema:
   a. Created 3 Tables (Event, Participant and EventParticipant):

```
CREATE TABLE Event (
eventId INT PRIMARY KEY,
eventName VARCHAR(255),
eventDateTime DATETIME,
totalSpaces INT
);

CREATE TABLE Participant (
pId INT PRIMARY KEY,
surName VARCHAR(255),
givenName VARCHAR(255),
);

CREATE TABLE EventParticipant (
eventId INT,
pId INT,
PRIMARY KEY (eventId, pId),
FOREIGN KEY (eventId) REFERENCES Event(eventId) ON DELETE
CASCADE
FOREIGN KEY (pId) REFERENCES Participant(pId) ON DELETE CASCADE
);
```

Populating tables with the data:

```
CREATE TABLE TempEventData(
eventId INT,
eventName VARCHAR(255),
eventDateTime DATETIME,
totalSpaces INT,
pId INT,
surName VARCHAR(255),
givenName VARCHAR(255)
);

LOAD DATA LOCAL INFILE '/home/odnerindheim/Downloads/data.txt'
INTO TABLE TempEventData
FIELDS TERMINATED BY ';'
LINES TERMINATED BY '\n'
```

Odne Rindheim
With corrected task 4.1, 6.3 and 6.5

(eventId, eventName, eventDateTime, totalSpaces, pId, surname, givenName);

INSERT INTO Event (eventId, eventName, eventDateTime, totalSpaces)
SELECT DISTINCT eventId, eventName, eventDateTime, totalSpaces
FROM TempEventData;

INSERT INTO Participant (pId, surname, givenName)
SELECT DISTINCT pId, surname, givenName FROM TempEventData;

INSERT INTO EventParticipant (eventID, pId)
SELECT DISCINCT eventId, pId FROM TempEventData;

Checked to see all was correct with:
SELECT * FROM Event
SELECT * FROM Participant
SELECT * FROM EventParticipant

Clean up:
DROP TABLE TempEventData;

2. Queries
    a. Find the maximum number of events that a single participant will attend.
        i. SELECT MAX(event_count) AS max_events
           FROM (SELECT COUNT(*) AS event_count
           FROM EventParticipant
           GROUP BY pId) AS max_events_count;

           Returned: 4.
    b. List names of participants that attend the largest number of events.
        i. SELECT pId, givenName, surname
           FROM Participant
           WHERE pId IN (
           SELECT pID
           FROM EventParticipant
           GROUP BY pId
           HAVING COUNT(*) = (
           SELECT MAX(event_count)
           FROM (SELECT COUNT(*) AS event_count
           FROM EventParticipant
           GROUP BY pId) AS max_events_count
)
           );

           Returned 3 rows:
           Astrid Gjerstad, Une Hov, Liselotte Martinussen.
    c. What events are attended by Ludvid Rustad?

Odne Rindheim
With corrected task 4.1, 6.3 and 6.5

i. SELECT eventName
FROM Event
WHERE eventId IN (
SELECT eventId
FROM EventParticipant
WHERE pId = (
SELECT pId
FROM Participant
WHERE givenName = ‘Ludvig’ AND surname = ‘Rustad’
)
);

Returned 3 rows:
Knarvikmila Lyderhorn opp, Stones at Koengen

d. List participants that attend exactly 3 events.
   i. SELECT pId, givenName, surname
   FROM Participant
   WHERE pId IN (
   SELECT pId
   FROM EventParticipant
   GROUP BY pId
   HAVING COUNT(*) = 3
   );

   Returned 89 rows.

e. What events do they attend, the participants that attend 3 events?
   i. SELECT DISTINCT e.eventId, e.eventName
   FROM Participant p
   INNER JOIN (
   SELECT pId, COUNT(*) AS event_count
   FROM EventParticipant
   GROUP BY pId
   HAVING event_count = 3
) AS p_count ON p.pId = p_count.pId
   INNER JOIN EventParticipant ep ON p.pId = ep.pId
   INNER JOIN Event e ON ep.eventId = e.eventId;

f. Are there any events that is not attended by anybody that attends three events?
   i. SELECT eventId, eventName
   FROM Event
   WHERE eventId NOT IN (
   SELECT DISCTINCT eventId
   FROM EventParticipant
   WHERE pId IN (
   SELECT pId
   FROM EventParticipant
   GROUP BY pId

HAVING COUNT(*) = 3
)
);

Returns 1 row:
5, Guided tour to løvstakken.

g. Are there any events that is attended only by the participants that attend only one event, i.e. participants attending more than one event, is there an event that none of them attend, but attended by those attending one event?
    i. SELECT eventId, eventName
       FROM Event
       WHERE eventId IN (
       SELECT DISTINCT eventId
       FROM EventParticipant
       WHERE pId IN (
       SELECT pId
       FROM EventParticipant
       GROUP BY pId
       HAVING COUNT(*) = 1
)
) AND eventId NOT IN (
       SELECT DISTINCT eventId
       FROM EventParticipant
       WHERE pId IN (
       SELECT pId
       FROM EventParticipant
       GROUP BY pId
       HAVING COUNT(*) > 1
)
       );

       Returned 1 row:
       5, Guided tour to løvstakken.

3. Making procedure takeSpace:
    a. DELIMITER //
       CREATE PROCEDURE takeSpace (
       IN pId_param INT,
       IN eventId_param INT
)
       BEGIN
       DECLARE available_spaces INT;
       START TRANSACTION;
       SELECT totalSpaces – COUNT(ep.pId) INTO available_spaces
       FROM Event e
       LEFT JOIN EventParticipant ep ON e.eventId = ep.eventId

Odne Rindheim
With corrected task 4.1, 6.3 and 6.5

```
WHERE e.eventId = eventId_param
GROUP BY e.eventId;
FOR UPDATE;

IF available_spaces > 0 THEN
INSERT INTO EventParticipant (eventId, pId)
VALUES (eventId_param, pId_param);
SELECT CONCAT('space booked for participant ', pId_param, ' at event ',
eventId_param) AS message;
COMMIT;
ELSE
SELECT 'Event is sold out' AS message;
COMMIT;
END IF;
END //
DELIMITER ;
```

4. SET profiling = 1;

   CALL takeSpace (4234474, 3);

   SET profiling = 0;

   SHOW PROFILES;

   In my query it is "Query_ID 7" which takes the longest time which is:

   ```
   SELECT totalSpace – COUNT (ep.pId) INTO available_spaces
   FROM Event e
   LEFT JOIN EventParticipant ep ON e.eventId = ep.eventId
   WHERE e.evendId = eventId_param
   GROUP BY e.eventId;
   ```

   When multiple concurrent bookings for the same event occur, it introduces possibility of contention for resources within the database system. This conention can lead to increased latency and potential performance deg due to following factors: Locking, Resource Utilization, Transaction Isolation and Deadlocks.
   To mitigate the impact, you can use strategies to avoid like: Optimistic concurrency control, database sharding, caching and optimized indexing.
5. To reduce time consumption and avoid the need for explicit locks, one potential denormalization scheme involves adding a 'bookedSpaces' column to the event table, by denormalizing Event table, we eliminate the need for JOIN and GROUP BY, which will speed up queries and reduce contention.

   ALTER TABLE Event ADD COLUMN bookedSpaces INT DEFAULT 0;

   DELIMITER //

Odne Rindheim
With corrected task 4.1, 6.3 and 6.5

```
CREATE PROCEDURE takeSpace (
IN pId_param INT,
IN eventId_param INT
)
BEGIN
DECLARE available_spaces INT;
START TRANSACTION;
SELECT totalSpaces – bookedSpaces INTO available_spaces
FROM Event
WHERE eventId = eventId_param;
FOR UPDATE;
IF available_spaces > 0 THEN
UPDATE Event
SET bookedSpaces = bookedSpaces + 1
WHERE eventId = eventId_param;
INSERT INTO EventParticipant (eventId, pId)
VALUES (eventId_param, pId_param);
SELECT CONCAT (...message..) AS Message;
COMMIT;
ELSE
SELECT 'Event sold out' AS Message;
COMMIT;
END IF;
END //
DELIMITER ;

SET profiling = 1;
CALL takeSpace(2906681, 7);
SET profiling = 0;
SHOW PROFILES;
```

This query took a total of 0.0169 seconds, meanwhile the other query took 0.013seconds.. While this query did take longer, it is generally better to not have JOIN operations and GROUP BY clauses, as these are more computationally intensive.