A2+A3

Type	Homework

Assignment 2

Total Price Calculator Trigger

```
CREATE FUNCTION get_student_age(s_name VARCHAR(30))
RETURNS INTEGER AS $$
DECLARE
    student_age INTEGER;
BEGIN
    SELECT EXTRACT(YEAR FROM AGE(s.s_dob)) INTO student_age
    FROM student s
    WHERE s.s_username = s_name;

    RETURN student_age;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION get_price_multiplier(s_username VARCHAR(30))
RETURNS NUMERIC(5, 2) AS $$
DECLARE
    student_age INTEGER;
BEGIN
    student_age := get_student_age(s_username);

    RETURN (
        SELECT price
        FROM pricetype
        WHERE p_type = (
            CASE
                WHEN student_age < 10 THEN 'CHILD'
                WHEN student_age < 15 THEN 'TEEN1'
                WHEN student_age < 20 THEN 'TEEN2'
                ELSE 'ADULT'
            END
        )
    );
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION calculate_total_price()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.unit_price IS NULL THEN
        NEW.total_price := NULL;
    ELSE
        NEW.total_price := NEW.quantity * NEW.unit_price * get_price_multiplier(NEW.s_username);
    END IF;

    RETURN NEW;
END;
```

```
$$ LANGUAGE plpgsql;


CREATE TRIGGER set_total_price
BEFORE INSERT OR UPDATE ON lpurchase
FOR EACH ROW EXECUTE FUNCTION calculate_total_price();
```

The `set_total_price` trigger in PostgreSQL is executed **before every** `INSERT` **or** `UPDATE` **operation** on the `lpurchase` table. It calls the `calculate_total_price()` function, which calculates the `total_price` for a purchase. If `unit_price` is `NULL`, `total_price` remains `NULL`; otherwise, it is computed as `quantity * unit_price * get_price_multiplier(NEW.s_username)`. The `get_price_multiplier` function determines a pricing factor based on the student's age, retrieved via the `get_student_age` function, which calculates the student's age from their date of birth (`s_dob`). This setup ensures that total pricing dynamically adjusts based on the student's age category (e.g., **CHILD, TEEN1, TEEN2, ADULT**) before inserting or updating purchase records.

Referential Integrity

Insert the following row into table lesson:

```
INSERT INTO public.lesson(
t_id, s_username, subjectcode)
VALUES (1, 'Wolf', 'CH');
```

What error do you get and why?

Answer:

```
ERROR:  null value in column "lesson_time" of relation "lesson" violates not-null constraint
Failing row contains (1, null, Wolf, null, CH).

SQL state: 23502
Detail: Failing row contains (1, null, Wolf, null, CH).
```

We get this error, because `INSERT INTO` statement does not include a value for the `lesson_time` column. However, the table definition for `lesson` has a **NOT NULL constraint** on `lesson_time`. This means that every inserted row **must** have a value for `lesson_time`.

What happens when we insert the following two rows into table lesson?

```
INSERT INTO lesson (t_id, lesson_time, s_username, subjectcode)
 VALUES (14, '2024-04-08 05:22:12.000000', 'Mickey', 'EN');

INSERT INTO lesson (t_id, lesson_time, s_username, subjectcode)
 VALUES (14, '2024-04-08 05:22:12.000000', 'Rose', 'EN');
```

Does the database react as we expect?

Answer:

	t_id integer	lesson_time timestamp without time zone	s_username character varying (30)	insertion_time timestamp without time zone	subjectcode character (2)
2	1	2024-01-19 18:46:19.879467	Rose	2025-02-18 17:59:30.777694	[null]
3	14	2024-01-31 18:55:18.210004	Rose	2025-02-18 17:59:30.777694	[null]
4	14	2024-01-31 18:55:07.066989	Mickey	2025-02-18 17:59:30.777694	[null]
5	14	2023-04-06 05:22:12	Mickey	2025-02-18 17:59:30.777694	[null]
6	1	2023-03-06 05:22:12	Mickey	2025-02-18 17:59:30.777694	[null]
7	14	2024-04-14 18:55:01.706886	Rose	2025-02-18 17:59:30.777694	[null]
8	2	2024-04-14 18:55:01.706886	Mickey	2025-02-18 17:59:30.777694	[null]
9	1	2024-02-07 05:22:12	Mickey	2025-02-18 18:01:28.230769	[null]
10	1	2024-03-07 05:22:12	Rose	2025-02-18 18:04:36.577597	[null]
11	14	2024-04-08 05:22:12	Mickey	[null]	EN
12	14	2024-04-08 05:22:12	Rose	[null]	EN

they were both inserted successfully into the database.

Assignment 3

PostGreSQL Transaction Example

Run the following command sequence, in PSQL:

```
START TRANSACTION;
 INSERT into test values (1,'Miller');
 INSERT into test values (2,'Doe');
 ROLLBACK;
START TRANSACTION;
 INSERT into test values (3,'Neuer');
 INSERT into test  values (4,'Herrmann');
 COMMIT;
 INSERT into test values (5,'Niemann');
 INSERT into test  values (6,'Smith');
 rollback;
```

Answer:

Data Output Messages Notifications

	ID [PK] integer	name character varying (30)
1	3	Neuer
2	4	Herrmann
3	5	Niemann
4	6	Smith

because of rollback, first two inserts are undone, then the following to rows are committed so they are inserted in the table, two rows are inserted: `(5, 'Niemann')` and `(6, 'Smith')`. since there was no explicit `START TRANSACTION` before these inserts, they are

treated as auto committed. The `ROLLBACK` has no effect because the previous inserts were already committed automatically.

---

run the same command sequence in PGAdmin4:

```
START TRANSACTION;
 INSERT into test values (1,'Miller');
 INSERT into test values (2,'Doe');
 ROLLBACK;
START TRANSACTION;
 INSERT into test values (3,'Neuer');
 INSERT into test  values (4,'Herrmann');
 COMMIT;
 INSERT into test values (5,'Niemann');
 INSERT into test  values (6,'Smith');
 rollback;
```

Which rows are in the table after running the sequence?

Answer:

	ID [PK] integer	name character varying (30)
1	3	Neuer
2	4	Herrmann

The **two inserts (** `Miller` **and** `Doe` **) are part of the same transaction**. `ROLLBACK;` undoes all the changes made within the transaction, so the table remains empty after this.

Then the second transaction starts **(** `Neuer` **and** `Herrmann` **)** are part of this transaction. `COMMIT;` permanently saves these changes in the database, so now the table contains these two entries.

These two inserts (`Niemann` and `Smith`) not part of the explicit transaction. When `ROLLBACK;` is executed, it does not affect the previous `INSERT` operations because they were already committed automatically. The rollback has no effect on the already-inserted rows.

Set auto commit OFF in command line and Run the same command sequence again on command line:

```
START TRANSACTION;
INSERT into test values (1,'Miller');
INSERT into test values (2,'Doe');
ROLLBACK;
START TRANSACTION;
INSERT into test values (3,'Neuer');
INSERT into test  values (4,'Herrmann');
COMMIT;
INSERT into test values (5,'Niemann');
INSERT into test  values (6,'Smith');
rollback;
```

Which rows are in the table after running the same sequence in PSQL but with auto commit off?

Answer:

Data Output | Messages | Notifications

	ID [PK] integer	name character varying (30)
1	3	Neuer
2	4	Herrmann

since auto commit was turned off , the last to rows won't be committed and thus when rollback is executed, they will not be inserted in the table.

Run the same command sequence again on command line – but without the last rollback command!

```
START TRANSACTION;
INSERT into test values (1,'Miller');
INSERT into test values (2,'Doe');
ROLLBACK;
START TRANSACTION;
INSERT into test values (3,'Neuer');
INSERT into test  values (4,'Herrmann');
COMMIT;
INSERT into test values (5,'Niemann');
INSERT into test  values (6,'Smith');
```

Check that transaction is still active:

```
SELECT * FROM pg_stat_activity WHERE state IN ('active', 'idle in transaction');
```

```
agency=*#  SELECT * FROM pg_stat_activity WHERE state IN ('active', 'idle in transaction');
 datid | datname | pid  | leader_pid | usesysid | usename  | application_name | client_addr | client_hostname | client_p
ort |        backend_start         |          xact_start          |         query_start          |         state_chang
e          | wait_event_type | wait_event | state  | backend_xid | backend_xmin | query_id |
          query                                              |  backend_type
-------+---------+------+------------+----------+----------+------------------+-------------+-----------------+---------
----+------------------------------+------------------------------+------------------------------+---------------------
-----------+-----------------+------------+--------+-------------+--------------+----------+----------------------------
-----------------------------------------------------+----------------
 16388 | agency  | 3884 |            |       10 | postgres | psql             | ::1         |                 |       58
051 | 2025-03-12 12:26:39.05162+04 | 2025-03-12 12:32:13.571846+04 | 2025-03-12 12:33:53.718998+04 | 2025-03-12 12:33:53
.719004+04 |                 |            | active |         789 |          789 |          | SELECT * FROM pg_stat_activ
ity WHERE state IN ('active', 'idle in transaction'); | client backend
```

end transaction manually by typing `command commit;` in PSQL.
Check, which rows are in the table:

Answer:

Data Output | Messages | Notifications

	ID [PK] integer	name character varying (30)
1	3	Neuer
2	4	Herrmann

Transactions on Different Rows

Default Isolation level

```
-- T1
BEGIN;
   SELECT t_id, t_payment FROM teacher WHERE t_id = 5;
   SELECT pg_sleep(15);
   UPDATE teacher SET t_payment = 0 WHERE t_id = 5;
COMMIT;

-- T2
BEGIN;
   SELECT t_id, t_payment FROM teacher WHERE t_id = 6 FOR UPDATE;
   SELECT pg_sleep(15);
   UPDATE teacher SET t_payment = t_payment + 3 WHERE t_id = 6;
COMMIT;
```

Answer:

```
agency=# SELECT t_id, t_payment FROM teacher WHERE t_id in (5, 6);
 t_id | t_payment
------+-----------
    5 |      0.00
    6 |     16.00
(2 rows)
```

both transactions ran through and commit since they modify different rows.

Repeatable read

```
-- T1
BEGIN ISOLATION LEVEL REPEATABLE READ;
   SELECT t_id, t_payment FROM teacher WHERE t_id = 5;
   SELECT pg_sleep(15);
   UPDATE teacher SET t_payment = 0 WHERE t_id = 5;
COMMIT;

-- T2
BEGIN ISOLATION LEVEL REPEATABLE READ;
   SELECT t_id, t_payment FROM teacher WHERE t_id = 6 FOR UPDATE;
   SELECT pg_sleep(15);
```

```
    UPDATE teacher SET t_payment = t_payment + 3 WHERE t_id = 6;
COMMIT;
```

Answer:

```
agency=#  SELECT t_id, t_payment FROM teacher WHERE t_id in (5, 6);
 t_id | t_payment
------+-----------
    5 |      0.00
    6 |     19.00
(2 rows)
```

Both transactions run through and commit since they modify different rows and repeatable read ensures that all rows inside transaction are not modified by other one.

What conclusions do you draw regarding PostgreSQL isolation mechanism?

PostgreSQL MVCC ensures high concurrency by allowing transactions on different rows to proceed without blocking. Read Committed (default) prevents dirty reads but allows others, Repeatable Read prevents non-repeatable reads, and Serializable ensures full isolation at the cost of potential rollbacks. Its row-level locking and versioning optimize performance while maintaining ACID compliance.

```
-- T1
BEGIN;
   SELECT t_payment FROM teacher WHERE t_id = 6;
   SELECT pg_sleep(15);
   UPDATE teacher SET t_payment = 0 WHERE t_id = 6;
COMMIT;

-- T2
   BEGIN;
      UPDATE teacher SET t_education = 'Bachelor' WHERE t_id = 6;
   COMMIT;
```

Answer:

```
agency=# SELECT t_id, t_education, t_payment FROM teacher WHERE t_id = 6;
 t_id | t_education | t_payment
------+-------------+-----------
    6 | Bachelor    |      0.00
(1 row)
```

both transactions ran through and commit.

Repeatable read

```
-- T1
BEGIN ISOLATION LEVEL REPEATABLE READ;
   SELECT t_payment FROM teacher WHERE t_id = 6;
   SELECT pg_sleep(15);
   UPDATE teacher SET t_payment = 0 WHERE t_id = 6;
COMMIT;

-- T2
BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
   UPDATE teacher SET t_education = 'Bachelor' WHERE t_id = 6;
COMMIT;
```

Answer:

```
agency=#  SELECT t_id, t_education, t_payment FROM teacher WHERE t_id = 6;
 t_id | t_education | t_payment
------+-------------+-----------
    6 | Bachelor    |      0.00
(1 row)
```

Since second transaction tries to update same row which is locked by first transaction. Second transaction commits, but the first one rolls back.

Concurrent Write Transactions

```
-- T1
BEGIN;
   SELECT t_payment FROM teacher WHERE t_id = 1;
   SELECT pg_sleep(15);
   UPDATE teacher SET t_payment = 0 WHERE t_id = 1;
COMMIT;

-- T2
BEGIN;
   UPDATE teacher SET t_payment = t_payment + 3 WHERE t_id = 1;
   SELECT t_payment FROM teacher WHERE t_id = 1;
   SELECT pg_sleep(15);
COMMIT;
```

Answer:

```
agency=# SELECT t_id, t_payment FROM teacher WHERE t_id = 1;
 t_id | t_payment
------+-----------
    1 |      0.00
(1 row)
```

First transaction 1 selects and gets suspended, releases lock. Then transaction 2 increments payment by 3 and releases lock. Finally first transaction sets payment to 0.

Isolation Levels

1. Which different isolation levels does the SQL standard have?

The SQL standard (SQL-92) defines **four transaction isolation levels**, each determining how transactions interact and handle concurrent access.

Isolation Level	Prevents Dirty Reads?	Prevents Non-Repeatable Reads?	Prevents Phantom Reads?
Read Uncommitted	❌ No	❌ No	❌ No
Read Committed	✅ Yes	❌ No	❌ No
Repeatable Read	✅ Yes	✅ Yes	❌ No
Serializable	✅ Yes	✅ Yes	✅ Yes

---

2. What anomalies (consistency problems) do the different levels prevent?

Anomalies occur when transactions interact in an undesired way, leading to data inconsistencies. The isolation levels control how these anomalies are prevented.

1. Dirty Reads

- **Issue:** A transaction reads **uncommitted** changes from another transaction.
- **Example:**

```
-- Transaction 1 (T1)
UPDATE accounts SET balance = 500 WHERE id = 1;

-- Transaction 2 (T2) (before T1 commits)
SELECT balance FROM accounts WHERE id = 1; -- Sees 500 (dirty read)

-- T1 rolls back, T2 has incorrect data.
```

- **Prevented by:** Read Committed, Repeatable Read, Serializable.
- **Allowed in:** Read Uncommitted.

2. Non-Repeatable Reads

- **Issue:** A transaction reads the same row twice, but another transaction modifies it between reads.
- **Example:**

```
-- Transaction 1 (T1)
SELECT balance FROM accounts WHERE id = 1; -- Reads 100

-- Transaction 2 (T2)
UPDATE accounts SET balance = 200 WHERE id = 1; COMMIT;

-- Transaction 1 (T1) reads again
SELECT balance FROM accounts WHERE id = 1; -- Now sees 200 (non-repeatable read)
```

- **Prevented by:** Repeatable Read, Serializable.
- **Allowed in:** Read Uncommitted, Read Committed.

3. Phantom Reads

- **Issue:** A transaction re-executes a query and gets **different** results because another transaction inserted or deleted rows.
- **Example:**

```
-- Transaction 1 (T1)
SELECT COUNT(*) FROM orders WHERE amount > 1000; -- Returns 5

-- Transaction 2 (T2)
INSERT INTO orders (id, amount) VALUES (101, 1500); COMMIT;

-- Transaction 1 (T1) re-executes
SELECT COUNT(*) FROM orders WHERE amount > 1000; -- Now returns 6 (phantom read)
```

- **Prevented by:** Serializable.
- **Allowed in:** Read Uncommitted, Read Committed, Repeatable Read.

---

3. Which isolation levels does Postgres offer?

PostgreSQL supports **four** SQL-standard isolation levels:

Isolation Level	Prevents Dirty Reads?	Prevents Non-Repeatable Reads?	Prevents Phantom Reads?
Read Uncommitted	❌ No (but behaves like Read Committed)	❌ No	❌ No
Read Committed	✅ Yes	❌ No	❌ No
Repeatable Read	✅ Yes	✅ Yes	❌ No
Serializable	✅ Yes	✅ Yes	✅ Yes

Key Notes for PostgreSQL:

- **Read Uncommitted** behaves the same as **Read Committed** (dirty reads are not actually allowed).
- **Repeatable Read** prevents non-repeatable reads but **does not prevent phantom reads**.
- **Serializable** ensures complete isolation using **Serializable Snapshot Isolation (SSI)**.

---

4. Which isolation levels does MySQL / MariaDB offer?

MySQL (using **InnoDB**) and MariaDB support **four isolation levels** similar to the SQL standard:

Isolation Level	Prevents Dirty Reads?	Prevents Non-Repeatable Reads?	Prevents Phantom Reads?
Read Uncommitted	❌ No	❌ No	❌ No
Read Committed	✅ Yes	❌ No	❌ No
Repeatable Read (Default)	✅ Yes	✅ Yes	❌ No (but prevents in some cases)
Serializable	✅ Yes	✅ Yes	✅ Yes

Key Notes for MySQL/MariaDB:

- **Default isolation level in MySQL is Repeatable Read.**
- **Gap Locks:** In **Repeatable Read**, MySQL uses **gap locks** (next-key locking) to **prevent some phantom reads**, but it **does not fully prevent them**.
- **Serializable mode uses table-level locking**, reducing concurrency.
