Information systems lab assignment 2

Erik Bijl (s2581582)
Emilio Oldenziel (s2509679)
Group 5

November 2018

Tasks

We will describe how we solved the tasks that were given in the assignment.

1 Normalization

We normalize the following schema up to the third normal form:

customer [**<u>custno</u>**, cust_name, cust_addr, cust_phone, (artist_id, artist_name, art_code, art_title, pur_date, price)]

1.1 First Normal Form

The first normal form must satisfy the rule that a relation must contain only atomic values at each row and column. Therefore, it is not allowed to to have multiple arts in the same row as in the schema above. We create a new row for each art that a customer has bought. Also we assume that the rest of the fields could have at most one value. Some interesting assumptions include that a customer can have only one address, one phone number, one art and one artist that created this art. The schema is given in Listing 1 and shown in Figure 1. Note, that the convention is used that the primary key is underlined and bold printed where a foreign key is only underlined.

customer [**<u>custno</u>**, cust_name, cust_addr, cust_phone, artist_id, artist_name, art_code, art_title, pur_date, price]

1.2 Second Normal Form

The second normal form must satisfy the rule that every non-key attribute is fully functionally dependent on the primary key. Therefore, we split the fields

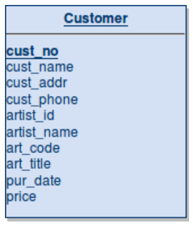

Figure 1: The first normal form

which results in three tables that of customer, art and purchase. Important to note is that in the purchase table also the pur_date is part of the primary key. This is because a customer could buy the same art multiple times when he also sells it afterwards.

customer [**custno**, cust_name, cust_addr, cust_phone]
art[**art_code**, art_title, artist_id, artist_name]
purchase[**cust_no, art_code, pur_date**, price]

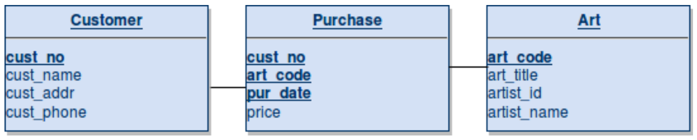

Figure 2: The second normal form

1.3 Third Normal Form

The third normal form must satisfy the following rule there is no transitive functional dependency between non-key attributes. This property holds for the customer and purchase tables but we find such a relation in the art table. The artist_name is fully functional depended on the artist_id which violates the rule above. We separate this relation by creating an additional table which gives us the following schema shown in Figure 3:

customer [**custno**, cust_name, cust_addr, cust_phone]
purchase[**cust_no, art_code, pur_date**, price]
art[**art_code**, art_title, artist_id]
artist[**artist_id**, artist_name]

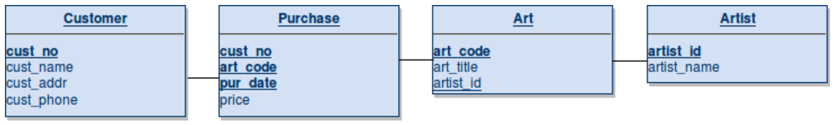

Figure 3: The third normal form

2 Triggers

To uppercase the an artist's and customer's name on an INSERT or UPDATE we implemented two triggers on the tables where they are part of. Both triggers call a procedure that applies the SQL uppercase function on the string value as shown in Listing 1.
As an additional check before inserting, it is checked whether the price is greater than zero. This is implemented as a trigger on the purchase table which contains the price. The procedure which is called by the purchase table trigger check if *price* $<= 0$. If this is the case, it raises an exception to indicate that the price is not positive, otherwise it returns the record.

3 Appendix

Listing 1: SQL Schema for creating the tables and triggers

```sql
/* create tables customer, artist, art and purchase */
CREATE TABLE customer (
     cust_no int,
     cust_name varchar(50),
     cust_addr varchar(200),
     cust_phone varchar(15),
     CONSTRAINT customers_pk PRIMARY KEY (cust_no)
 );

CREATE TABLE purchase (
     cust_no int,
     art_code varchar(200),
     pur_date DATE,
     price int,
     CONSTRAINT purchases_pk PRIMARY KEY (cust_no,
         art_code, pur_date)
 );

CREATE TABLE artist (
     artist_id int,
     artist_name varchar(50),
     CONSTRAINT artist_pk PRIMARY KEY (artist_id)
 );

CREATE TABLE art (
     art_code varchar(200),
     art_title varchar(200),
     artist_id int,
     CONSTRAINT art_pk PRIMARY KEY (art_code),
     FOREIGN KEY (artist_id) REFERENCES artist(artist_id)
);

/* function that changes a name of customer to its
    uppercase */
CREATE OR REPLACE FUNCTION cust_uppercase()
RETURNS TRIGGER AS
$BODY$
BEGIN
     UPDATE customer SET cust_name = UPPER(cust_name);
     RETURN NEW;
END;
$BODY$
language plpgsql;

/* function that changes a name of artist to its
    uppercase */
CREATE OR REPLACE FUNCTION art_uppercase()
RETURNS TRIGGER AS
$BODY$
BEGIN
     UPDATE artist SET artist_name = UPPER(artist_name);
     RETURN NEW;
END;
$BODY$
language plpgsql;

/* function that raises an error when price is not
    positive */
CREATE OR REPLACE FUNCTION price_pos()
RETURNS TRIGGER AS
$BODY$
BEGIN
  IF NEW.price <= 0 THEN
     RAISE EXCEPTION 'price␣is␣not␣positive';
  END IF;
  return new;
END;
$BODY$
```

```
language plpgsql;

/* trigger that is triggered after each insert or update
    on customer */
CREATE TRIGGER cust_upper
AFTER INSERT OR UPDATE
ON customer
FOR EACH ROW
WHEN (pg_trigger_depth() = 0)
EXECUTE PROCEDURE cust_uppercase();

/* trigger that is triggered after each insert or update
    on artist */
CREATE TRIGGER art_upper
AFTER INSERT OR UPDATE
ON artist
FOR EACH ROW
WHEN (pg_trigger_depth() = 0)
EXECUTE PROCEDURE art_uppercase();

/* trigger that is raises error when price is not greater
     than 0 */
CREATE TRIGGER price_check
AFTER INSERT OR UPDATE
ON purchase
FOR EACH ROW
WHEN (pg_trigger_depth() = 0)
EXECUTE PROCEDURE price_pos();
```