

## Contents

Term paper task.....	2
Database scheme.....	4
Database realization.....	4
Tables creation.....	4
Queries.....	6
CASE query.....	6
Multi-table VIEW.....	6
Query, with SELECT, FROM and WHERE subqueries.....	6
Correlated subqueries.....	7
Multitable HAVING query.....	8
ANY query.....	8
Indexes.....	8
Trigger.....	9
Insert, update, delete for all tables.....	10
Transaction.....	12
Cursor.....	13
Scalar function.....	13
Vector function.....	14
User roles.....	15
App realization.....	15
Connecting to database.....	15
Insert/update/delete.....	15
Summary.....	17
Literature and sources.....	17
Screenshots.....	18
Source files.....	21
Dbsqlqueries.cpp.....	21
Mainwindow.cpp.....	26

## Term paper task

Develop a client-server application, the server part of which must be implemented on Microsoft SQL Server or PostgreSQL, which is a domain model in accordance with the assignment option. Within a given subject area, implement a given (by option) scheme of relations, i.e. highlight the entities and their attributes so that the relationships between the entities correspond to the presented schema.

A slight deviation from the given scheme is allowed. As part of the course work, it is necessary to implement and use the following components on the server side when demonstrating the application:

1. Permanent tables and relationships between them, the number of tables and the presence of relationships should correspond to the task, it is allowed to increase the number of tables and their fields for a more adequate representation of the subject area;
  2. In the application (on the client side), implement at least five requests (to demonstrate work skills) that can implement the tasks from clause 3.
  3. Implement requests for tasks (in any fragments of scripts, both on the server side and on the client side):
    - a. Compound multi-table query with CASE expression;
    - b. Multi-table VIEW, with the ability to update it;
    - c. Queries that contain a subquery in the SELECT, FROM, and WHERE sections (at least one in each);
    - d. Correlated subqueries (minimum 3 queries).
    - e. A multi-table query containing grouping of records, aggregate functions, and a parameter used in the HAVING clause;
    - f. Queries containing ANY(SOME) or ALL predicate (for each predicate);
  4. Create indexes (at least 3 pieces) to increase the speed of query execution; Provide indexes of different types. Indexes must be created for different tables. Include a query plan in the report that shows how the index was applied when the query was executed.
  5. In the table (according to the option), provide a field that is filled (and updated) automatically when a trigger fires when adding, updating and deleting data, be able to demonstrate the trigger when the application is running. Triggers should process only those records that were added, changed or deleted during the current operation (transaction).
  6. Operations of adding, deleting and updating to implement in the form of stored procedures or functions with parameters for all tables;
  7. Implement a separate stored procedure or function consisting of several separate operations in the form of a single transaction, which, under certain conditions, can be committed or rolled back;
  8. Implement a cursor to update individual data (calculate the value of the fields of the selected table);
  9. Implement your own scalar and vector functions. Functions to save in the database; 10.
- Distribution of user rights: provide for at least two users with a different set of privileges. Each set of privileges is represented as a role.
- The client must ensure the addition, modification and deletion of data across the entire subject area. Adding, editing data in the table should be done in a separate window. It is forbidden to specify the values of primary and foreign keys as input data, including for linking tables - to ensure referential integrity, the user must select values from the directory, and the corresponding values must be substituted programmatically (in one way or another - automatically).

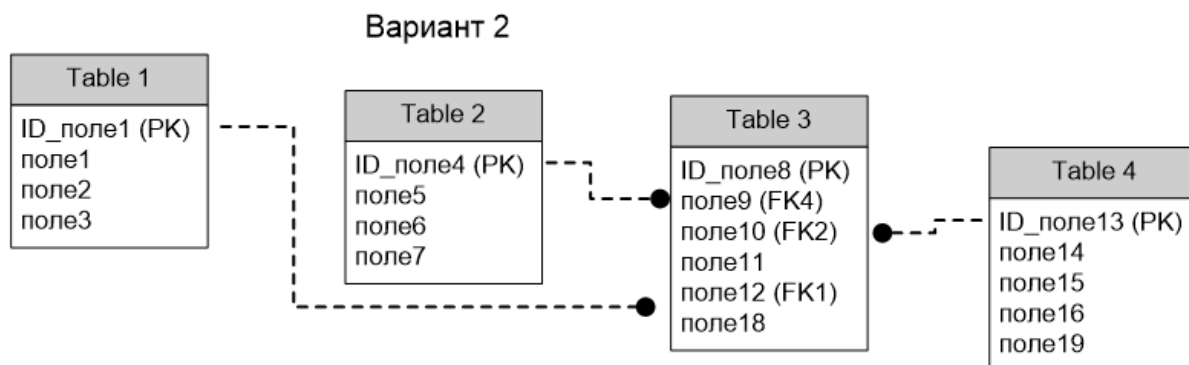
The client must be implemented on the means (programming languages) studied (studied) within the framework of the curriculum, including related disciplines.

The assignment variant (see table) determines the subject area and the nature of the relationship (a slight change in the nature of the relationship is allowed to ensure compliance with the variant relationship scheme, i.e. so that the tables “fit” into the scheme specified in the variant). An increase in relations (the number of tables and the number of fields of the given tables, triggers, etc.) is allowed for a more complete representation of the subject area (the set of tables and fields given in the variant determines the minimum amount for developing a term paper).

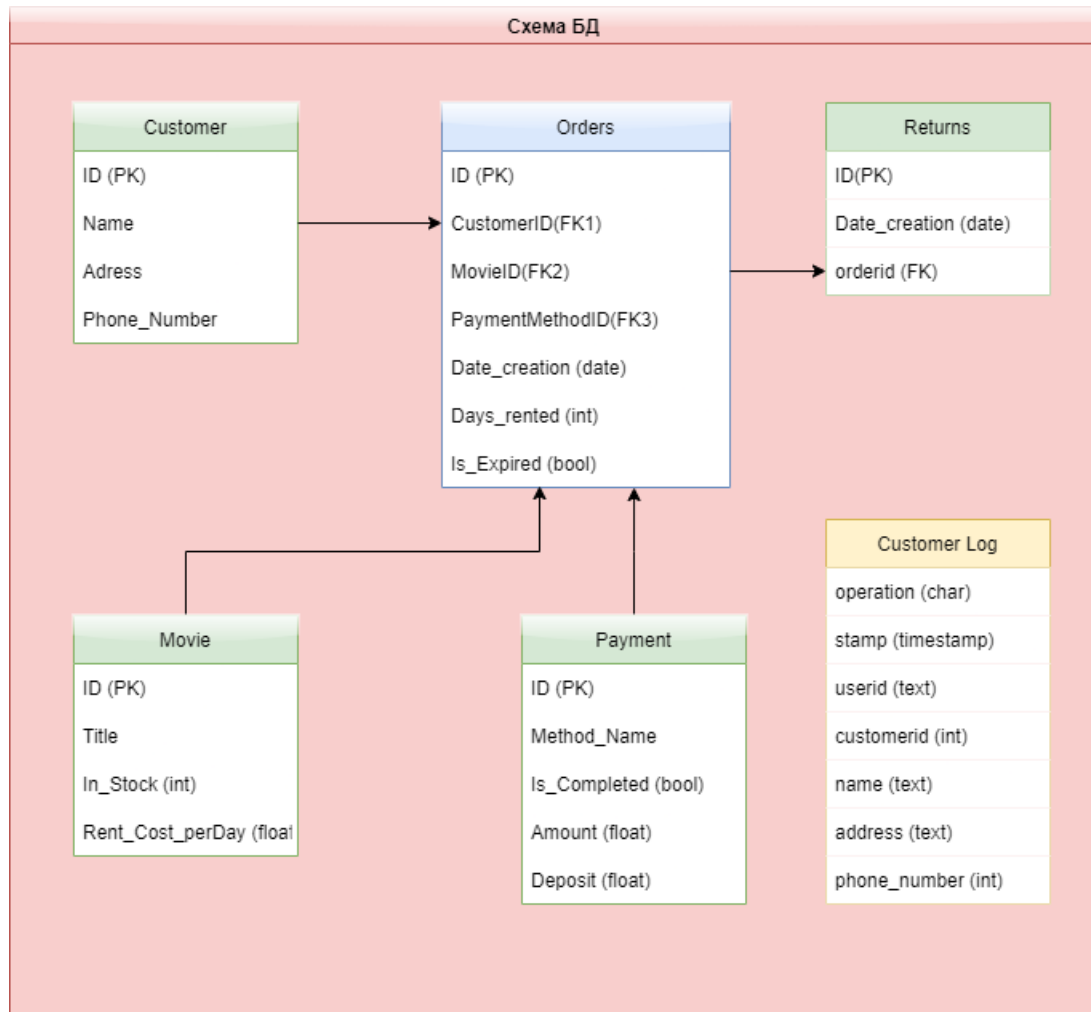
#### Variant: 79 (Movie rent sevrvice)

79	Ведение базы данных пункта проката видеопродукции	Ведение классификаторов и информационной базы по видеофильмам и их пользователям в пункте проката. Учет приема, выдачи видеофильмов, расчет размера оплаты.	2
----	---	---	---

#### Scheme



## Database scheme



## Database realization

### Tables creation

```
CREATE TABLE public.Customers(  
    CustomerID serial primary key NOT NULL,  
    Name text NOT NULL,  
    Adress text NOT NULL,  
    PhoneNumber int  
)
```

## Movie library table

```
1 CREATE TABLE public.Movies(  
2     MovieID serial primary key NOT NULL,  
3     Title text NOT NULL,  
4     Author text NOT NULL,  
5     RentCostperDay money  
6 )
```

## Payments

```
1 CREATE TABLE public.payments(  
2     PaymentID serial PRIMARY KEY NOT NULL,  
3     Method_name text NOT NULL,  
4     is_completed boolean,  
5     amount money NOT NULL,  
6     deposit money  
7 );
```

## Orders

```
1 CREATE TABLE public.ShopOrders(  
2     OrderID serial PRIMARY KEY NOT NULL,  
3     Date_creation date,  
4     Days_rented int,  
5     is_expired boolean default false,  
6  
7     fk_ShopOrders_customerid integer REFERENCES public.customers (customerid)  
8     ON UPDATE CASCADE ON DELETE CASCADE,  
9     fk_ShopOrders_movieid integer REFERENCES public.movies (movieid)  
10    ON UPDATE CASCADE ON DELETE CASCADE,  
11    fk_ShopOrders_paymentid integer REFERENCES public.payments (paymentid)  
12    ON UPDATE CASCADE ON DELETE CASCADE  
13 )
```

## Returns

```
CREATE TABLE public.ShopReturns(  
    ReturnID serial PRIMARY KEY NOT NULL,  
    Date_creation date,  
  
    fk_ShopReturns_orderid integer REFERENCES public.shoporders (orderid)  
    ON UPDATE CASCADE ON DELETE CASCADE  
)
```

# Queries

## CASE query

	customerid	adress	name	city
1	39	Chicago	Nick	Not NYC
2	2	Moscow	Dmitry	Not NYC
3	38	Spb	Vlad	Not NYC
4	1	NYC	John	From NYC

## Multi-table VIEW

```
SELECT customer.customerid,
CASE WHEN customer.adress = 'NYC' THEN 'From NYC'
FROM public.customers A
WHERE customerid = ANY (SELECT fk_shoporders_customerid FROM public.shoporders);
```

	orderid	date_creation	is_expired	shoporders_paymentid
1	3	15.06.2002	false	16
2	4	31.05.2022	false	17
3	5	02.06.2022	false	18

## Query, with SELECT, FROM and WHERE subqueries

```
1 SELECT rets.returnid,
2     (SELECT pay.is_completed
3      FROM payments pay
4      WHERE pay.paymentid IN (SELECT ord.fk_shoporders_paymentid
5      FROM shoporders ord
6      WHERE ord.orderid = rets.fk_shopreturns_orderid)) AS is_completed,
7     rets.date_creation
8 FROM shopreturns rets;
6 JOIN shopreturns shret ON orders.orderid = shret.fk_shopreturns_orderid
7 GROUP BY orders.orderid, orders.date_creation, orders.is_expired
8 ORDER BY orders.orderid;
```

	returnid [PK] integer	is_completed boolean	date_creation date
1	3	true	2002-06-15
2	4	true	2022-06-02
3	6	true	2022-06-08

## Correlated subqueries

1

```
SELECT
    pay.is_completed,
    (SELECT mov.title FROM public.movies AS mov WHERE mov.movieid = ord.fk_shoporders_movieid),
    ord.date_creation FROM public.shoporders AS ord
LEFT JOIN public.payments AS pay ON pay.paymentid = ord.fk_shoporders_paymentid;
```

2

```
SELECT
    ord.is_expired, cust.name,
    (SELECT ret.date_creation FROM public.shopreturns AS ret WHERE ret.fk_shopreturns_orderid = ord.orderid),
    ord.days_rented FROM public.shoporders AS ord
LEFT JOIN public.customers AS cust ON ord.fk_shoporders_customerid = cust.customerid;
```

3

```
1 CREATE OR REPLACE VIEW public.corrsubq3 AS
2 SELECT ord.orderid, pay.amount, (SELECT ret.date_creation
3     FROM shopreturns ret
4     WHERE ret.fk_shopreturns_orderid = ord.orderid) AS date_creation,
5     ord.days_rented FROM shoporders ord
6 LEFT JOIN payments pay ON ord.fk_shoporders_paymentid = pay.paymentid;
```

1 exm:

	is_completed	title	date_creation
1	true	Shawhenk	15.06.2002
2	true	2 Smoking ...	31.05.2022
3	true	Godfather	02.06.2022
4	true	Psycho	02.02.2002

2 exm:

	is_expired	name	date_creation	days_rented
1	false	John	15.06.2002	2
2	false	Dmitry	02.06.2022	3
3	false	Vlad	08.06.2022	5

3 exm:

	orderid integer	amount money	date_creation date	days_rented integer
1	3	\$4.00	2002-06-15	2
2	4	\$3.00	2022-06-02	3
3	5	\$15.00	2022-06-08	5

## Multitable HAVING query

```
1 SELECT (SELECT COUNT(orderid) FROM public.shoporders),
2 COUNT(customerid), adress
3 FROM public.customers
4 GROUP BY adress
5 HAVING COUNT(customerid) > 2;
```

	count	count	adress
1	4	3	Chicago
2	4	5	NYC
3	4	3	Moscow

## ANY query

```
SELECT orderid, date_creation
FROM public.shoporders AS ord
WHERE fk_shoporders_paymentid = ANY
      (SELECT paymentid
       FROM public.payments
       WHERE amount > 1::integer::money);
```

	orderid	date_creation
1	3	15.06.2002
2	4	31.05.2022
3	5	02.06.2022
4	6	02.02.2002

## Indexes

```
1 CREATE UNIQUE INDEX customer_index ON public.customers (customerid);
2 CREATE INDEX ON public.customers ((LOWER(name))); --регистронезависимость
3 CREATE INDEX CONCURRENTLY customer_blockoff_index ON public.customers (customerid);-- откл. блокировки записи
4
5 CREATE UNIQUE INDEX movie_index ON public.movies (movieid);
6 CREATE INDEX ON public.movies ((LOWER(title)));
7 CREATE INDEX ON public.movies ((LOWER(author)));
8 CREATE INDEX CONCURRENTLY movie_blockoff_index ON public.movies (movieid);S
9
10 CREATE UNIQUE INDEX payment_index ON public.payments (paymentid);
11 CREATE INDEX ON public.payments ((LOWER(method_name)));
12 CREATE INDEX CONCURRENTLY payment_blockoff_index ON public.payments (paymentid);|
```



```
1 SELECT FROM public.customers WHERE LOWER(name) = 'peter';
```

## Trigger

```
CREATE TRIGGER customer_log  
AFTER INSERT OR UPDATE OR DELETE ON  
public.customers  
FOR EACH ROW EXECUTE FUNCTION  
public.process_customer_log();
```

```
CREATE OR REPLACE FUNCTION public.process_customer_log()  
RETURNS TRIGGER AS $customer_log$  
BEGIN  
    IF(TG_OP = 'DELETE') THEN  
        INSERT INTO public.customer_log  
        SELECT 'D', now(), user, OLD.*;  
    ELSEIF(TG_OP = 'UPDATE') THEN  
        INSERT INTO public.customer_log  
        SELECT 'U', now(), user, NEW.*;  
    ELSEIF(TG_OP = 'INSERT') THEN  
        INSERT INTO public.customer_log  
        SELECT 'I', now(), user, NEW.*;  
    END IF;  
    RETURN NULL;  
END;  
$customer_log$ LANGUAGE plpgsql;
```

```
CREATE TABLE public.customer_log(  
    operation char(1) NOT NULL,  
    userid text NOT NULL,  
    customerid int NOT NULL,  
    name text,  
    adress text,  
    phone_number int,  
    stamp timestamp  
);
```

## Insert, update, delete for all tables

```
1 CREATE OR REPLACE FUNCTION public.add_customer(text, text, int)
2 RETURNS void AS $$
3 BEGIN
4     INSERT INTO public.customers(name, adress, phonenumber) VALUES ($1,$2,$3);
5 END;
6 $$ LANGUAGE plpgsql;
```

### delete\_customer(integer)

General	Definition	Code	Options	Parameters	Security	SQL
---------	------------	------	---------	------------	----------	-----

1						
2		BEGIN				
3		DELETE FROM public.customers where customerid = \$1;				
4		END;				

### update\_customer(integer, text, text, integer)



General	Definition	Code	Options	Parameters	Security	SQL
---------	------------	------	---------	------------	----------	-----

1						
2		BEGIN				
3		UPDATE public.customers SET name = \$2, adress = \$3, phonenumber = \$4 WHERE customerid = \$1;				
4		END;				

Query Editor	Query History
--------------	---------------

1	CREATE OR REPLACE FUNCTION public.add_movie(text, text, money)
2	RETURNS void AS \$\$
3	BEGIN
4	INSERT INTO public.movies(title, author, rentcostperday) VALUES(\$1,\$2,\$3);
5	END;
6	\$\$ LANGUAGE plpgsql;

```
1 CREATE OR REPLACE FUNCTION public.delete_movie(int)
2 RETURNS void AS $$
3 BEGIN
4     DELETE FROM public.movies where movieid = $1;
5 END;
6 $$ LANGUAGE plpgsql;
```

 `update_movie(integer, text, text, money)` 

General Definition Code Options Parameters Security SQL

```
1
2 BEGIN
3 UPDATE public.movies SET title = $2, author = $3, rentcostperday = $4 WHERE movieid = $1;
4 END;
```

 `delete_payment(integer)`

General Definition Code Options Parameters Security SQL



```
1
2 BEGIN
3 DELETE FROM public.payments where paymentid = $1;
4 END;
```

```
CREATE OR REPLACE FUNCTION public.add_order(date, int, boolean, int, int, int)
RETURNS void AS $$
BEGIN
    INSERT INTO public.orders(date_creation, days_rented, is_expired,
                              fk_shoporders_customerid, fk_shoporders_movieid,
                              fk_shoporders_paymentid)
    VALUES ($1, $2, $3, $4, $5, $6);
END;
$$ LANGUAGE plpgsql;
```

 `delete_order(integer)`

General Definition Code Options Parameters Security SQL

```
1
2 ▼ BEGIN
3 DELETE FROM public.shoporders WHERE orderid = $1;
```

 `add_return(date, integer)` 

General Definition Code Options Parameters Security SQL

```
1
2 ▼ BEGIN
3 INSERT INTO public.shopreturns(date_creation, fk_shopreturns_orderid)
4 VALUES ($1, $2);
5 END;
```

## delete\_return(integer)

General   Definition   Code   Options   Parameters   Security   SQL

```
1
2 ▼ BEGIN
3     DELETE FROM public.shopreturns WHERE returnid = $1;
```

### Transaction

```
CREATE OR REPLACE PROCEDURE
public.add_payment_transact(text, bool, money, money)
AS $$
BEGIN
INSERT INTO public.payments
(method_name, is_completed, amount, deposit)
VALUES
($1, $2, $3, $4);
IF(
    (SELECT is_completed FROM public.payments
    WHERE paymentid =
    currval('public.payments_paymentid_seq')) = 'false')
    THEN ROLLBACK;
END IF;
COMMIT;
END;
$$
LANGUAGE plpgsql;
```

## Cursor

```
1 CREATE OR REPLACE PROCEDURE public.cursor_delete_shopreturn_ondate(date)
2 AS $$
3 DECLARE
4     my_cursor CURSOR FOR SELECT date_creation FROM public.shopreturns;
5     temp_date date;
6 ▼ BEGIN
7     OPEN my_cursor;
8 ▼     LOOP
9         FETCH my_cursor INTO temp_date;
10        IF NOT FOUND THEN EXIT; END IF;
11 ▼        IF(temp_date = $1) THEN
12            DELETE FROM public.shopreturns WHERE CURRENT OF my_cursor;
13        END IF;
14    END LOOP;
15 END;
16 $$
17 LANGUAGE plpgsql;
```

## Scalar function

```
CREATE OR REPLACE FUNCTION public.total_orders()
    RETURNS integer AS
$$
DECLARE
    total integer;
BEGIN
    SELECT COUNT(*) INTO total FROM public.shoporders;
    RETURN total;
END;
$$
LANGUAGE plpgsql;
```

total_orders	
1	4

## Vector function

```
CREATE OR REPLACE FUNCTION public.orders_by_date(sdate date)
  RETURNS TABLE(
    orderid int, days_rented int, is_expired bool) AS
$$
  DECLARE
    rec record;
BEGIN
  FOR rec IN EXECUTE
    'SELECT orderid, days_rented, is_expired
  FROM public.shoporders WHERE date_creation = $1' USING sdate
  LOOP
    orderid = rec.orderid;
    days_rented = rec.days_rented;
    is_expired = rec.is_expired;
    RETURN NEXT;
  END LOOP;
END;
$$
LANGUAGE plpgsql;
```

Orders by date

2002-06-15

VECTOR func.

	orderid	days_rented	is_expired
1	3	2	false

## User roles

```
CREATE ROLE basicuser PASSWORD '111';

GRANT USAGE ON SCHEMA public to basicuser;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO basicuser;
GRANT EXECUTE ON FUNCTION public.add_customer TO basicuser;
GRANT EXECUTE ON FUNCTION public.add_movie TO basicuser;
GRANT EXECUTE ON FUNCTION public.add_payment TO basicuser;
GRANT EXECUTE ON FUNCTION public.add_order TO basicuser;
GRANT EXECUTE ON FUNCTION public.add_return TO basicuser;

1 CREATE ROLE super superuser login password '000';
2 GRANT EXECUTE ON ALL FUNCTIONS IN SCHEMA public to super;
```

## App realization

### Connecting to database

```
void LoginForm::on_pushButton_Login_clicked()
{
    loginUsername = ui->lineEdit_EnterUsername->text();
    loginPassword = ui->lineEdit_EnterPassword->text();

    this->close();
}

void MainWindow::on_pushButton_connectToDb_clicked()
{
    dbUsername = newLogin->loginUsername;
    dbPassword = newLogin->loginPassword;

    DB = QSqlDatabase::addDatabase("QPSQL", "QtConnection");
    DB.setHostName("localhost");
    DB.setDatabaseName("Kursova4sem");
    DB.setUserName(dbUsername);
    DB.setPassword(dbPassword);
    DB.setPort(5432);
    bool ok = DB.open(); Δ Value stored to 'ok' during its initialization is

    if(DB.open())
    {
        ui->label_ConnectionStatus->setText("Successful connection.");
        qDebug()<<"Database connection OPENED";
    }
    else
    {
        ui->label_ConnectionStatus->setText("Wrong username or password. Restart.");
        DB.close();
    }
}
```

### Insert/update/delete

```

 QSqlQueryModel *dbSqlQueries::loadCustomersFromDb(QSqlDatabase db)
{
    QSqlQueryModel *modelCustomers = new QSqlQueryModel();
    QSqlQuery *query = new QSqlQuery(db);

    query->prepare("SELECT * FROM public.Customers");//
    query->exec();
    modelCustomers->setQuery(std::move(*query));
    delete query;

    //qDebug() << (modelCustomers->rowCount());
    return modelCustomers;
}

void MainWindow::on_pushButton_LoadMovieLibrary_clicked()
{
    ui->tableView_Movies->setModel(newQueries.loadMovieLibraryFromDb(DB));
    qDebug()<<"Movie library LOADED";
}

```

Пример добавления, удаления и изменения данных:

```

void dbSqlQueries::addNewMovieToLibrary(QSqlDatabase db, QString mTitle, QString mAuthor, QString mCost)
{
    QSqlQuery *query = new QSqlQuery(db);
    query->exec("SELECT public.add_movie('"+ mTitle + "','"+ mAuthor + "','"+ mCost + "')");
}

void dbSqlQueries::deleteMovieFromLibrary(QSqlDatabase db, QTableView *tableView)
{
    QSqlQuery *query = new QSqlQuery(db);
    query->exec("SELECT public.delete_movie(" + FetchIDSelectedInTable(tableView) + ")");
}

void dbSqlQueries::updateMovieInLibrary(QSqlDatabase db, QTableView *tableView, QString mTitle, QString mAuthor, QString mCost)
{
    QString FetchedID = FetchIDSelectedInTable(tableView);
    QSqlQuery *query = new QSqlQuery(db);
    query->exec("SELECT public.update_movie('"+ FetchedID + "','"+ mTitle + "','"+ mAuthor + "','"+ mCost + "')");
}

```

В случае добавления нового заказа, необходимо выбрать записи из нескольких таблиц:

```

void dbSqlQueries::createNewOrder(QSqlDatabase db, QTableView *tableView_Customer,
                                   QTableView *tableView_Movie, QTableView *tableView_Payment,
                                   QString dateCreation, QString daysRented)
{
    QString FetchedCustomerID = FetchIDSelectedInTable(tableView_Customer);
    QString FetchedMovieID = FetchIDSelectedInTable(tableView_Movie);
    QString FetchedPaymentID = FetchIDSelectedInTable(tableView_Payment);

    QSqlQuery *query = new QSqlQuery(db);
    query->exec("SELECT public.add_order('"+dateCreation+"','"+daysRented+"','"+ "false" + "','"+FetchedCustomerID+"','"+FetchedMovieID+"")
}

```



```
QString dbSqlQueries::FetchIDSelectedInTable(QTableView *tableview)
{
    return QString("%1").arg(tableview
        ->model()
        ->data(tableview
        ->model()
        ->index(tableview->
        currentIndex().row(),0)).toInt());
}
```

Также для возврата фильма, нужно выбрать номер заказа, который требуется вернуть:

```
void dbSqlQueries::createNewReturn(QSqlDatabase db, QTableView *tableview_Orders, QString dateCreation)
{
    QString FetchedOrderID = FetchIDSelectedInTable(tableview_Orders);
    QSqlQuery *query = new QSqlQuery(db);
    query->exec("SELECT public.add_return('"+dateCreation+"','"+FetchedOrderID+"')");
}
```

В программе можно установить текущую дату:

```
void MainWindow::on_pushButton_setCurrentDate_clicked()
{
    int year = ui->dateEdit->date().year(),
        month = ui->dateEdit->date().month(),
        day = ui->dateEdit->date().day();

    currentProgramDate = QString::number(year) + "-"
        +QString::number(month)+ "-"
        +QString::number(day);
    qDebug()<<"Program date set:"<<currentProgramDate;
}
```

## Summary

В результате выполнения мною курсовой работы я разработал клиент-серверное приложения, реализованное с помощью PostgreSQL (в качестве серверной части) и фреймворка C++ Qt (в качестве клиентской части). Приложение-клиент реализует концепцию «Model-View-Controller» и позволяет удобно работать с базой данных.

В базе данных были предусмотрены следующие особенности:

- Индексы для увеличения скорости выполнения запросов;
- Триггеры;
- Операции добавления, изменения и удаления данных в виде функций;
- Собственные скалярная и векторная функции;
- Курсор на обновление данных;
- Распределение прав пользователей с разным набором привилегий.


## Literature and sources

- PostgreSQL : Документация  
URL: <https://postgrespro.ru/docs/postgresql/>
- Материалы по дисциплине — принципы, технологии и средства организации данных компонентов и программного обеспечения

- Qt Documentation: <https://doc.qt.io/>

## Addition

### Screenshots

 Login
 ✕

# LOGIN

Username:

Password:

Log in

Movie table

	movieid	title	author	rentc
1	1	Shawhenk	Dapramon	\$2.00
2	3	2 Smoking ...	Richi	\$1.00
3	13	Godfather	Coppola	\$3.00
4	16	Schindlers list	Spielberg	\$4.00
5	17	Casablanca	Curtiz	\$6.00
6	18	Psycho	Hitchcock	\$5.00
7	19	Forrest Gump	Zemeckis	\$1.00
8	20	Star wars 1977	Lucas	\$3.00
9	21	Apocalypse now	Coppola	\$6.00
10	22	Saving Private ...	Spielberg	\$7.00

Movie library control

Title

Author

Rent price (daily)

Add new movie

Update selected

Delete selected

Connection

CONNECT TO DATABASE

Successful connection.

Customers table

	customerid	name	adress	phon
1	1	John	NYC	499995
2	2	Dmitry	Moscow	866007
3	38	Vlad	Spb	788885
4	39	Nick	Chicago	788888
5	40	Smith	Tucson	198880
6	41	Kevin	Berlin	598888
7	42	Alexander	Novosibirsk	744466
8	43	Alfred	NYC	188995
9	44	Brian	NYC	122255
10	45	Peter	Madrid	499988

Customers control

Name

Adress

Phone number

Add new customer

Update info

Delete customer

Payments table

	paymentid	method_name	is_completed	amount
1	16	VISA7998	true	\$4.00
2	17	MCARD8990	true	\$3.00
3	18	Cash	true	\$15.00
4	19	VISA9909	true	\$5.00
5	20	MC5446	true	\$1.00

Payments control

Load payments

Payment method

Amount

Deposit

Update info

Add new payment

Delete payment

Payment completed

Orders

Create a new order

Return an order

31.05.2022

Today is:

Set

Queries

Select a movie to order:

	movieid	title	author
1	1	Shawhenk	Dapramon
2	3	2 Smoking ...	Richi
3	13	Godfather	Coppola
4	16	Schindlers list	Spielberg
5	17	Casablanca	Curtiz
6	18	Psycho	Hitchcock

Select a customer:

	customerid	name	adress	phc
1	1	John	NYC	4999
2	2	Dmitry	Moscow	8660
3	38	Vlad	Spb	7888
4	39	Nick	Chicago	7888
5	40	Smith	Tucson	1988
6	41	Kevin	Berlin	5988

Number of days: 3

Payment details

Method:

Amount:

☒ Completed

Deposit:

Confirm payment

	paymentid	method_name	is_completed	
1	16	VISA7998	true	\$4.00
2	17	MCARD8990	true	\$3.00
3	18	Cash	true	\$15.00
4	19	VISA9909	true	\$5.00

Finish order

## All orders

	orderid	date_creation	days_rented	is_expired	toporders_custom	shoporders_movi
1	3	15.06.2002	2	false	1	1
2	4	31.05.2022	3	false	2	3
3	5	02.06.2022	5	false	38	13
4	6	02.02.2002	1	false	39	18

## All returns

	returnid	date_creation	shopreturns_orde
1	3	15.06.2002	3
2	4	02.06.2022	4
3	6	08.06.2022	5

Return selected order

Delete selected order

Delete selected return

## Dialog

	customerid	adress	name	city
1	39	Chicago	Nick	Not NYC
2	2	Moscow	Dmitry	Not NYC
3	38	Spb	Vlad	Not NYC
4	1	NYC	John	From NYC

CASE-query

Subquery

Correlated subq

1

2

3

HAVING

Orders by date

Multi table VIEW

ANY ALL query

SCALAR Func.

VECTOR func.

## Source files

### Dbsqlqueries.cpp

```
#include "dbsqlqueries.h"
dbSqlQueries::dbSqlQueries()
{

}

QString dbSqlQueries::FetchIDSelectedInTable(QTableView *tableview)
{
    return QString("%1").arg(tableview
        ->model()
        ->data(tableview
        ->model()
        ->index(tableview->
        currentIndex().row(),0)).toInt());
}

 QSqlQueryModel* dbSqlQueries::loadMovieLibraryFromDb(QSqlDatabase db)
{

    QSqlQueryModel *modelMovies = new QSqlQueryModel();
    QSqlQuery *query = new QSqlQuery(db);

    query->prepare("SELECT * FROM public.Movies");
    query->exec();
    modelMovies->setQuery(std::move(*query));
    delete query;

    //qDebug() << (modelMovies->rowCount());
    return modelMovies;
}

void dbSqlQueries::addNewMovieToLibrary(QSqlDatabase db, QString mTitle, QString mAuthor, QString mCost)
{
    QSqlQuery *query = new QSqlQuery(db);
    query->exec("SELECT public.add_movie('"+ mTitle + "','"+ mAuthor + "','"+ mCost + "')");
}

void dbSqlQueries::deleteMovieFromLibrary(QSqlDatabase db, QTableView *tableview)
{
    QSqlQuery *query = new QSqlQuery(db);
    query->exec("SELECT public.delete_movie('"+ FetchIDSelectedInTable(tableview) + "')");
}

void dbSqlQueries::updateMovieInLibrary(QSqlDatabase db, QTableView *tableview, QString mTitle, QString mAuthor, QString mCost)
{
    QString FetchedID = FetchIDSelectedInTable(tableview);
    QSqlQuery *query = new QSqlQuery(db);
    query->exec("SELECT public.update_movie('"+ FetchedID + "','"+ mTitle + "','"+ mAuthor + "','"+ mCost + "')");
}

float dbSqlQueries::selectMoviePrice(QSqlDatabase db, QTableView *tableview)
{
    float priceValue;
    QString FetchedID = FetchIDSelectedInTable(tableview);
    //qDebug() << "MovieID"<<FetchedID;

    QSqlQuery *query = new QSqlQuery(db);
    query->prepare("SELECT rentcostperday FROM public.Movies WHERE movieid = :movieid");
    query->bindValue(":movieid", FetchedID);
    query->exec();
    qDebug()<<query->size();

    query->next();
    priceValue = query->value(0).toFloat();

    //qDebug() <<"Selected price:"<< priceValue;
```

```

        return priceValue;
    }

    QSqlQueryModel *dbSqlQueries::loadCustomersFromDb(QSqlDatabase db)
    {
        QSqlQueryModel *modelCustomers = new QSqlQueryModel();
        QSqlQuery *query = new QSqlQuery(db);

        query->prepare("SELECT * FROM public.Customers");//
        query->exec();
        modelCustomers->setQuery(std::move(*query));
        delete query;

        //qDebug() << (modelCustomers->rowCount());
        return modelCustomers;
    }

    void dbSqlQueries::addNewCustomerToDb(QSqlDatabase db, QString cName, QString cAdress, QString
cPhoneNumber)
    {
        QSqlQuery *queryAddC = new QSqlQuery(db);

        queryAddC->exec("SELECT public.add_customer('"+ cName + "','"+ cAdress + "','"+ cPhoneNumber
+ "','");");
    }

    void dbSqlQueries::deleteCustomerFromDb(QSqlDatabase db, QTableView *tableview)
    {
        QSqlQuery *query = new QSqlQuery(db);
        query->exec("SELECT public.delete_customer("+ FetchIDSelectedInTable(tableview) + ");");
    }

    void dbSqlQueries::updateCustomerInDb(QSqlDatabase db, QTableView *tableview, QString cName, QString
cAdress, QString cPhoneNumber)
    {
        QString FetchedID = FetchIDSelectedInTable(tableview);
        QSqlQuery *query = new QSqlQuery(db);
        query->exec("SELECT public.update_customer('"+ FetchedID + "','"+ cName + "','"+ cAdress + "','"+
cPhoneNumber + "')");
    }

    QSqlQueryModel *dbSqlQueries::loadPaymentsFromDb(QSqlDatabase db)
    {
        QSqlQueryModel *modelPayments = new QSqlQueryModel();
        QSqlQuery *query = new QSqlQuery(db);

        query->prepare("SELECT * FROM public.Payments");//
        query->exec();
        modelPayments->setQuery(std::move(*query));
        delete query;

        //qDebug() << (modelPayments->rowCount());
        return modelPayments;
    }

    void dbSqlQueries::addNewPaymentToDb(QSqlDatabase db, QString pMethodName, bool pIsCompleted,
QString pAmount, QString pDeposit)
    {
        QSqlQuery *queryAddC = new QSqlQuery(db);
        QString pIsCompletedString;
        if(pIsCompleted == true){pIsCompletedString = "true";}
        else{pIsCompletedString = "false";}

        queryAddC->exec("SELECT public.add_payment('"+ pMethodName + "','"+ pIsCompletedString + "','"+
pAmount + "','"+ pDeposit + "')");
    }

    void dbSqlQueries::addNewPaymentToDbTRANSACTION(QSqlDatabase db, QString pMethodName, bool
pIsCompleted, QString pAmount, QString pDeposit)
    {
        QSqlQuery *queryAddC = new QSqlQuery(db);

```

```

        QString pIsCompletedString;
        if(pIsCompleted == true){pIsCompletedString = "true";}
        else{pIsCompletedString = "false";}

        queryAddC->exec(QString("CALL
public.add_payment_transact('%1','%2','%3','%4');").arg(pMethodName).arg(pIsCompleted).arg(pAmount).
arg(pDeposit));

    }

void dbSqlQueries::deletePaymentFromDb(QSqlDatabase db, QTableView *tableview)
{
    QSqlQuery *query = new QSqlQuery(db);
    query->exec("SELECT public.delete_payment("+ FetchIDSelectedInTable(tableview)+");");
}

void dbSqlQueries::updatePaymentInDb(QSqlDatabase db, QTableView *tableview, QString pMethodName,
bool pIsCompleted, QString pAmount, QString pDeposit)
{
    QString FetchedID = FetchIDSelectedInTable(tableview);
    QString pIsCompletedString;
    if(pIsCompleted == true){pIsCompletedString = "true";}
    else{pIsCompletedString = "false";}
    QSqlQuery *query = new QSqlQuery(db);
    query->exec("SELECT public.update_payment('"+ FetchedID +"','"+ pMethodName +"','"+
pIsCompletedString +"','"+ pAmount +"','"+ pDeposit +"')");
}

void dbSqlQueries::createNewOrder(QSqlDatabase db, QTableView *tableview_Customer,
QTableView *tableview_Movie, QTableView *tableview_Payment,
QString dateCreation, QString daysRented)
{
    QString FetchedCustomerID = FetchIDSelectedInTable(tableview_Customer);
    QString FetchedMovieID = FetchIDSelectedInTable(tableview_Movie);
    QString FetchedPaymentID = FetchIDSelectedInTable(tableview_Payment);

    QSqlQuery *query = new QSqlQuery(db);
    query->exec("SELECT public.add_order('"+dateCreation+"','"+daysRented+"','"+ "false"
+"','"+FetchedCustomerID+"','"+FetchedMovieID+"','"+FetchedPaymentID+"')");
}

void dbSqlQueries::deleteOrderFromDb(QSqlDatabase db, QTableView *tableview)
{
    QSqlQuery *query = new QSqlQuery(db);
    query->exec("SELECT public.delete_order("+ FetchIDSelectedInTable(tableview)+");");
}

QSqlQueryModel *dbSqlQueries::loadOrdersFromDb(QSqlDatabase db)
{
    QSqlQueryModel *modelOrders = new QSqlQueryModel();
    QSqlQuery *query = new QSqlQuery(db);

    query->prepare("SELECT * FROM public.shoporders");//
    query->exec();
    modelOrders->setQuery(std::move(*query));
    delete query;

    return modelOrders;
}

QSqlQueryModel *dbSqlQueries::loadReturnsFromDb(QSqlDatabase db)
{
    QSqlQueryModel *modelReturns = new QSqlQueryModel();
    QSqlQuery *query = new QSqlQuery(db);

    query->prepare("SELECT * FROM public.shopreturns");//
    query->exec();
    modelReturns->setQuery(std::move(*query));
    delete query;

    return modelReturns;
}

```

```

}

void dbSqlQueries::createNewReturn(QSqlDatabase db, QTableView *tableview_Orders, QString
dateCreation)
{
    QString FetchedOrderID = FetchIDSelectedInTable(tableview_Orders);
    QSqlQuery *query = new QSqlQuery(db);
    query->exec("SELECT public.add_return('"+dateCreation+"','"+FetchedOrderID+"')");
}

void dbSqlQueries::deleteReturnFromDb(QSqlDatabase db, QTableView *tableview)
{
    QSqlQuery *query = new QSqlQuery(db);
    query->exec("SELECT public.delete_return('"+ FetchIDSelectedInTable(tableview)+"')");
}

QSqlQueryModel *dbSqlQueries::queryCASE(QSqlDatabase db)
{
    QSqlQueryModel *modelCASE = new QSqlQueryModel();
    QSqlQuery *query = new QSqlQuery(db);

    query->exec("SELECT * FROM public.casequeryview;");
    modelCASE->setQuery(std::move(*query));
    delete query;

    return modelCASE;
}

QSqlQueryModel *dbSqlQueries::queryMultitableVIEW(QSqlDatabase db)
{
    QSqlQueryModel *modelVIEW = new QSqlQueryModel();
    QSqlQuery *query = new QSqlQuery(db);

    query->exec("SELECT * FROM public.multitableview;");
    modelVIEW->setQuery(std::move(*query));
    delete query;

    return modelVIEW;
}

QSqlQueryModel *dbSqlQueries::querySubquerySelect(QSqlDatabase db)
{
    QSqlQueryModel *modelVIEW = new QSqlQueryModel();
    QSqlQuery *query = new QSqlQuery(db);

    query->exec("SELECT * FROM public.subqueryselect;");
    modelVIEW->setQuery(std::move(*query));
    delete query;

    return modelVIEW;
}

QSqlQueryModel *dbSqlQueries::queryCorrelatedSQ1(QSqlDatabase db)
{
    QSqlQueryModel *modelVIEW = new QSqlQueryModel();
    QSqlQuery *query = new QSqlQuery(db);

    query->exec("SELECT * FROM public.corrsqb1;");
    modelVIEW->setQuery(std::move(*query));
    delete query;

    return modelVIEW;
}

QSqlQueryModel *dbSqlQueries::queryCorrelatedSQ2(QSqlDatabase db)
{
    QSqlQueryModel *modelVIEW = new QSqlQueryModel();
    QSqlQuery *query = new QSqlQuery(db);

    query->exec("SELECT * FROM public.corrsqb2;");
    modelVIEW->setQuery(std::move(*query));
}

```



```

        delete query;

        return modelVIEW;
    }

    QSqlQueryModel *dbSqlQueries::queryCorrelatedSQ3(QSqlDatabase db)
    {
        QSqlQueryModel *modelVIEW = new QSqlQueryModel();
        QSqlQuery *query = new QSqlQuery(db);

        query->exec("SELECT * FROM public.corrsqb3;");
        modelVIEW->setQuery(std::move(*query));
        delete query;

        return modelVIEW;
    }

    QSqlQueryModel *dbSqlQueries::queryAnyAllView(QSqlDatabase db)
    {
        QSqlQueryModel *modelVIEW = new QSqlQueryModel();
        QSqlQuery *query = new QSqlQuery(db);

        query->exec("SELECT * FROM public.antalview;");
        modelVIEW->setQuery(std::move(*query));
        delete query;

        return modelVIEW;
    }

    QSqlQueryModel *dbSqlQueries::queryHaving(QSqlDatabase db)
    {
        QSqlQueryModel *modelVIEW = new QSqlQueryModel();
        QSqlQuery *query = new QSqlQuery(db);

        query->exec("SELECT (SELECT COUNT(orderid) FROM public.shoporders), COUNT(customerid), adress
FROM public.customers GROUP BY adress HAVING COUNT(customerid) > 2;");
        modelVIEW->setQuery(std::move(*query));
        delete query;

        return modelVIEW;
    }

    QSqlQueryModel *dbSqlQueries::queryScalarFunc_TotalOrders(QSqlDatabase db)
    {
        QSqlQueryModel *modelVIEW = new QSqlQueryModel();
        QSqlQuery *query = new QSqlQuery(db);

        query->exec("SELECT * FROM public.total_orders();");
        modelVIEW->setQuery(std::move(*query));
        delete query;

        return modelVIEW;
    }

    QSqlQueryModel *dbSqlQueries::queryVectorFunc_OrdersByDate(QSqlDatabase db, QString date)
    {
        QSqlQueryModel *modelVIEW = new QSqlQueryModel();
        QSqlQuery *query = new QSqlQuery(db);

        query->exec(QString("SELECT * FROM public.orders_by_date('%1')").arg(date));
        modelVIEW->setQuery(std::move(*query));
        delete query;

        return modelVIEW;
    }
}

```

## Mainwindow.cpp

```
void MainWindow::on_pushButton_connectToDb_clicked()
{
    dbUsername = newLogin->loginUsername;
    dbPassword = newLogin->loginPassword;

    DB = QSqlDatabase::addDatabase("QPSQL", "QtConnection");
    DB.setHostName("localhost");
    DB.setDatabaseName("Kursova4sem");
    DB.setUserName(dbUsername);
    DB.setPassword(dbPassword);
    DB.setPort(5432);
    bool ok = DB.open();

    if(DB.open())
    {
        ui->label_ConnectionStatus->setText("Successful connection.");
        qDebug()<<"Database connection OPENED";
    }
    else
    {
        ui->label_ConnectionStatus->setText("Wrong username or password. Restart.");
        DB.close();
    }
}

void MainWindow::on_pushButton_setCurrentDate_clicked()
{
    int year = ui->dateEdit->date().year(),
        month = ui->dateEdit->date().month(),
        day = ui->dateEdit->date().day();

    currentProgramDate = QString::number(year) + "-"
        +QString::number(month)+ "-"
        +QString::number(day);
    qDebug()<<"Program date set:"<<currentProgramDate;
}
}
```