

МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

**«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»**

Институт компьютерных наук и кибербезопасности
Высшая школа технологий искусственного интеллекта
Направление 02.03.01 Математика и компьютерные науки

Отчет по лабораторным работам

по дисциплине «Методы проектирования баз данных»

Обучающийся: _____

Шихалев А.О.

Руководитель: _____

Попов С.Г.

«_____» _____ 20__ г.

Санкт-Петербург, 2024

Содержание

Введение	3
1 Постановка задачи	4
2 Лабораторные работы	5
2.1 Создание представления	5
2.2 Событийная модель, триггеры	8
2.3 Разграничение прав доступа	15
2.4 Функции и процедуры	18
2.4.1 Функция	18
2.4.2 Процедура	19
2.4.2.1 Первый пример	21
2.4.2.2 Второй пример	23
2.4.2.3 Третий пример	24
2.5 Транзакции	25
Заключение	27
Список литературы	28

Введение

В данном отчёте описан результат выполнения лабораторных работ, которые расширяют функциональные возможности базы данных «Процесс прохождения кастинга претендентом на роль в фильме», которая была разработана в течение предыдущего семестрового курса «Теоретические основы баз данных».

1 Постановка задачи

Необходимо выполнить 5 лабораторных работ:

1. Создать представление, инкапсулирующее запрос. Продемонстрировать невозможность модификации представления, написать запрос, использующий в себе это представление.
2. Создать таблицу подсчета количества заявок каждого актера. Написать триггеры, автоматизирующие сбор статистики в таблице.
3. Создать двух пользователей: первый имеет доступ только на просмотр представления из первого задания, второй также может редактировать таблицы, участвующие в запросе представления.
4. Процедуры и функции.
5. Управление транзакциями.

2 Лабораторные работы

2.1 Создание представления

Задача: Разработать представление для хранения запроса внутри СУБД.

Формулировка запроса: Для каждого режиссера подсчитать общее количество фильмов, которые он срежиссировал, а также количество актеров, которые снимались в этих фильмах.

В результате реализации задачи было создано представление, хранящее в себе запрос, подсчитывающий для каждого режиссера общее количество снятых им фильмов и число актеров, которые снимались в этих фильмах. Код создания представления представлен в [листинге 1](#).

Листинг 1. Создание представления

```
1 CREATE VIEW director_view AS
2     SELECT d.id_director, d.name, d.surname,
3     COUNT(DISTINCT f.id_film) AS film_count,
4     COUNT(DISTINCT act.id_actor) AS actor_count
5     FROM director AS d
6     JOIN film AS f ON d.id_director = f.id_director
7     JOIN role AS r ON f.id_film = r.id_film
8     JOIN application AS app ON r.id_role = app.id_role
9     JOIN actor AS act ON app.id_actor = act.id_actor
10    GROUP BY d.id_director, d.name, d.surname;
```

Получившееся представление (view) является виртуальной таблицей со следующими атрибутами:

- id_director - id режиссера;
- d.name - имя режиссера;
- d.surname - фамилия режиссера;
- film_count - количество фильмов, снятых каждым режиссером;
- actor_count - количество актеров, подавших заявки на роль в фильмах каждого режиссера.

Результат вывода представления представлен на [рисунке 1](#).

При попытке модификации таблицы выводятся ошибки, поскольку представления не являются обновляемыми таблицами. Результат применения команд языка DML к созданному представлению представлен на [рисунке 2](#).

```
MySQL localhost:33060+ ssl casting SQL > select * from director_view limit 20;
```

id_director	name	surname	film_count	actor_count
1	Bill	Condon	3	190
2	Makoto	Shinkai	6	273
3	Кира	Муратова	6	411
4	Георгий	Юнгвальд-Хилькевич	6	322
5	Yimou	Zhang	6	300
6	Brothers	Coen	2	119
7	Larry	Clark	6	384
8	David	O. Russell	6	404
9	Steven	Spielberg	6	337
10	Александр	Прошкин	6	347
11	Jia	Zhangke	6	377
12	Curtis	Hanson	4	251
13	Billy	Wilder	5	302
14	Susanne	Bier	6	321
15	Ulrich	Seidl	4	219
16	John	Madden	6	320
17	Johnnie	To	6	292
18	Jean-Luc	Godard	6	357
19	Tim	Burton	6	438
20	Céline	Sciamma	6	372

Рис. 1. Вывод представления

```
MySQL localhost:33060+ ssl casting SQL > INSERT INTO director_view (id_director, name, surname, film_count, actor_count)
-> VALUES (10, 'Steven', 'Spielberg', 5, 30);
ERROR: 1471: The target table director_view of the INSERT is not insertable-into
MySQL localhost:33060+ ssl casting SQL > UPDATE director_view
-> SET
-> name = 'Steven Allan',
-> surname = 'Spielberg',
-> film_count = 6,
-> actor_count = 35
-> WHERE id_director = 10;
ERROR: 1288: The target table director_view of the UPDATE is not updatable
MySQL localhost:33060+ ssl casting SQL > DELETE FROM director_view
-> WHERE id_director = 1;
ERROR: 1288: The target table director_view of the DELETE is not updatable
```

Рис. 2. Ошибки, возникающие при попытке модификации представления

Задача: Написать запрос, включающий в себя созданное представление.

Формулировка запроса №1: Получить общее количество жанров и ролей в фильмах каждого режиссера.

В результате реализации задачи был написан запрос, представленный в [листинге 2](#).

Листинг 2. Код запроса, использующего представление

```
1 SELECT dv.id_director, dv.name, dv.surname, dv.film_count, dv.actor_count,
2     COUNT(DISTINCT gnr.id_genres) AS count_genres,
3     COUNT(DISTINCT role.id_role) AS count_role
4 FROM director_view as dv
5 JOIN film AS f ON dv.id_director = f.id_director
6 JOIN film_genres AS fg ON f.id_film = fg.id_film
7 JOIN genres AS gnr ON fg.id_genres = gnr.id_genres
8 JOIN role AS role ON f.id_film = role.id_film
9 GROUP BY dv.id_director, dv.name, dv.surname;
```

Результат выполнения запроса №1 представлен на [рисунке 3](#).

```
MySQL localhost:33060+ ssl casting SQL > SELECT dv.id_director, dv.name, dv.surname, dv.film_count, dv.actor_count,
-> COUNT(DISTINCT gnr.id_genres) AS count_genres,
-> COUNT(DISTINCT role.id_role) AS count_role
-> FROM director_view as dv
-> JOIN film AS f ON dv.id_director = f.id_director
-> JOIN film_genres AS fg ON f.id_film = fg.id_film
-> JOIN genres AS gnr ON fg.id_genres = gnr.id_genres
-> JOIN role AS role ON f.id_film = role.id_film
-> GROUP BY dv.id_director, dv.name, dv.surname;
```

id_director	name	surname	film_count	actor_count	count_genres	count_role
1	Bill	Condon	3	190	8	38
2	Makoto	Shinkai	6	273	14	54
3	Кира	Муратова	6	411	15	84
4	Георгий	Юнгвальд-Хилькевич	6	322	10	68
5	Yimou	Zhang	6	300	9	65
6	Brothers	Coen	2	119	6	22
7	Larry	Clark	6	384	11	79
8	David	O. Russell	6	404	14	85
9	Steven	Spielberg	6	337	9	73
10	Александр	Провкин	6	347	14	74
11	Jia	Zhangke	6	377	12	78
12	Curtis	Hanson	4	251	6	45
13	Billy	Wilder	5	302	10	66
14	Susanne	Bier	6	321	14	69
15	Ulrich	Seidl	4	219	11	42
16	John	Madden	6	320	14	70
17	Johnnie	To	6	292	13	61
18	Jean-Luc	Godard	6	357	11	81
19	Tim	Burton	6	438	14	85
20	Céline	Sciamma	6	372	10	81

Рис. 3. Результат выполнения запроса №1

Формулировка запроса №2: Получить количество утвержденных заявок для фильмов каждого режиссера.

В результате реализации задачи был написан запрос, представленный в [листинге 3](#).

Листинг 3. Код запроса, использующего представление

```
1 SELECT dv.id_director, dv.name, dv.surname, dv.film_count, dv.actor_count,
2     COUNT(CASE WHEN du.getting_a_role = 'Роль получена' THEN 1 END) AS
   ↳ count_received_roles
3 FROM director_view as dv
4 JOIN film AS f ON dv.id_director = f.id_director
5 JOIN role AS role ON f.id_film = role.id_film
6 JOIN application AS app ON role.id_role = app.id_role
7 JOIN doubles_audition AS du ON app.id_application = du.id_application
8 GROUP BY dv.id_director, dv.name, dv.surname
9 ORDER BY dv.id_director ASC;
```

Результат выполнения запроса №2 представлен на [рисунке 4](#).

```
MySQL localhost:33060+ ssl casting SQL > SELECT dv.id_director, dv.name, dv.surname, dv.film_count, dv.actor_count,
-> COUNT(CASE WHEN du.getting_a_role = 'Роль получена' THEN 1 END) AS count_received_roles
-> FROM director_view as dv
-> JOIN film AS f ON dv.id_director = f.id_director
-> JOIN role AS role ON f.id_film = role.id_film
-> JOIN application AS app ON role.id_role = app.id_role
-> JOIN doubles_audition AS du ON app.id_application = du.id_application
-> GROUP BY dv.id_director, dv.name, dv.surname
-> ORDER BY dv.id_director ASC;
```

id_director	name	surname	film_count	actor_count	count_received_roles
1	Bill	Condon	3	190	17
2	Makoto	Shinkai	6	273	22
3	Кира	Муратова	6	411	29
4	Георгий	Юнгвальд-Хилькевич	6	322	21
5	Yimou	Zhang	6	300	21
6	Brothers	Coen	2	119	11
7	Larry	Clark	6	384	34
8	David	O. Russell	6	404	28
9	Steven	Spielberg	6	337	32
10	Александр	Провкин	6	347	18
11	Jia	Zhangke	6	377	25
12	Curtis	Hanson	4	251	24
13	Billy	Wilder	5	302	14
14	Susanne	Bier	6	321	28
15	Ulrich	Seidl	4	219	14
16	John	Madden	6	320	27
17	Johnnie	To	6	292	15
18	Jean-Luc	Godard	6	357	24
19	Tim	Burton	6	438	27
20	Céline	Sciamma	6	372	34
21	Андрей	Звягинцев	4	250	16
22	Michael	Mann	6	349	34
23	Fritz	Lang	2	136	7
24	Michel	Gondry	6	303	22
25	Maren	Ade	6	385	34
26	Mélanie	Laurent	3	224	14
27	Don	Bluth	6	413	39
28	John	Carpenter	5	273	24
29	Jonathan	Demme	6	362	34
30	Steven	Soderbergh	6	410	29
31	Raymond	De Felitta	1	32	1

Рис. 4. Результат выполнения запроса №2

2.2 Событийная модель, триггеры

Задача: Создать таблицу подсчета количества заявок, поданных каждым актером. Для реализованной таблицы написать триггеры, автоматизирующие сбор статистики в таблице.

В результате реализации задачи была создана таблица **actor_apps**. Код создания таблицы представлен в [листинге 4](#).

Листинг 4. Код создания таблицы actor_apps

```
1 CREATE TABLE IF NOT EXISTS actor_apps(
2     id_actor int unsigned NOT NULL,
3     surname varchar(30) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
4     name varchar(20) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
5     patronymic varchar(20) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
6     num_apps int unsigned NOT NULL,
7     PRIMARY KEY(id_actor)
8 )ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

Полученная таблица была заполнена данными при помощи запроса, представленного в [листинге 5](#).

Листинг 5. Код заполнения таблицы actor_apps


```

1 INSERT INTO actor_apps(id_actor, surname, name, patronymic, num_apps)
2 SELECT a.id_actor, a.surname, a.name, a.patronymic,
3        COUNT(DISTINCT app.id_application) AS num_apps
4 FROM actor AS a
5 JOIN application AS app ON a.id_actor = app.id_actor
6 GROUP BY a.id_actor;

```

Результат заполнения таблицы actor_apps представлен на [рисунке 5](#).

id_actor	surname	name	patronymic	num_apps
1	Богдан	Арсений	Вячеславович	8
2	Гладков	Игорь	Андреевич	6
3	Анненкова	Виктория	Константиновна	5
4	Валиулин	Игорь	Русланович	5
5	Жумагали	Канат	Еркинович	6
6	Парфенова	Виктория	Данииловна	5
7	Булгаков	Арсений	Денисович	5
8	Александров	Марк	Егорович	6
9	Голубева	Валерия	Ярославовна	7
10	Гаврилова	Юлия	Владимировна	5
11	Попова	Алёна	Егоровна	6
12	Иванов	Лев	Константинович	5
13	Коротков	Антон	Юрьевич	6
14	Леонов	Фёдор	Платонович	5
15	Горюнова	София	Николаевна	5
16	Макеев	Иван	Никитич	6
17	Панкратов	Святослав	Александрович	7
18	Степанова	Елизавета	Ильинична	6
19	Сорокина	Александра	Николаевна	5
20	Наумов	Даниил	Кириллович	6
21	Шестакова	Анастасия	Семёновна	5
22	Гусев	Андрей	Григорьевич	5
23	Федорова	Мария	Степановна	6
24	Соловьева	Дарья	Андреевна	7
25	Родионова	Мария	Александровна	7
26	Беляев	Глеб	Матвеевич	6
27	Толкачев	Иван	Викторович	5
28	Емельянов	Николай	Макарович	5
29	Фролова	Василиса	Евгеньевна	6

Рис. 5. Результат заполнения таблицы actor_apps

После создания таблицы были созданы триггеры:

1. Триггер add_app

- Событие: Добавление новой заявки.
- Действие: Обновление числа заявок в таблице actor_apps.

- Принцип работы: при добавлении новой заявки в таблицу application срабатывает триггер, обновляющий количество заявок для соответствующего актера в таблице actor_apps.

Код триггера представлен в [листинге 6](#).

Листинг 6. Код триггера add_app

```

1 CREATE TRIGGER add_app
2 AFTER INSERT ON application
3 FOR EACH ROW
4 BEGIN
5     UPDATE actor_apps
6     SET num_apps = num_apps + 1
7     WHERE id_actor = NEW.id_actor;
8 END;

```

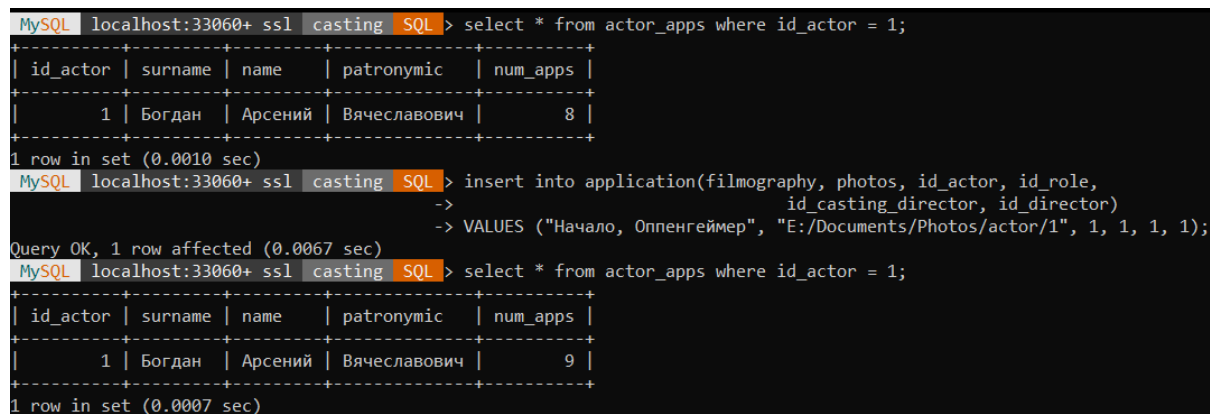
Для демонстрации работы данного триггера добавим новую заявку актеру с id_actor = 1 следующим образом:

```

1 INSERT INTO application (filmography, photos, id_actor, id_role,
2                          id_casting_director, id_director)
3 VALUES ("Начало, Оппенгеймер", "E:/Documents/Photos/actor/1", 1, 1, 1, 1);

```

Результат работы триггера представлен на [рисунке 6](#).



```

MySQL localhost:33060+ ssl casting SQL > select * from actor_apps where id_actor = 1;
+-----+-----+-----+-----+-----+
| id_actor | surname | name   | patronymic | num_apps |
+-----+-----+-----+-----+-----+
| 1       | Богдан  | Арсений | Вячеславович | 8       |
+-----+-----+-----+-----+-----+
1 row in set (0.0010 sec)

MySQL localhost:33060+ ssl casting SQL > insert into application(filmography, photos, id_actor, id_role,
->                          id_casting_director, id_director)
-> VALUES ("Начало, Оппенгеймер", "E:/Documents/Photos/actor/1", 1, 1, 1, 1);
Query OK, 1 row affected (0.0067 sec)

MySQL localhost:33060+ ssl casting SQL > select * from actor_apps where id_actor = 1;
+-----+-----+-----+-----+-----+
| id_actor | surname | name   | patronymic | num_apps |
+-----+-----+-----+-----+-----+
| 1       | Богдан  | Арсений | Вячеславович | 9       |
+-----+-----+-----+-----+-----+
1 row in set (0.0007 sec)

```

Рис. 6. Результат работы триггера add_app

2. Триггер delete_app

- Событие: Удаление заявки.
- Действие: Обновление числа заявок в таблице actor_apps.
- Принцип работы: при удалении заявки в таблице application срабатывает триггер, обновляющий количество заявок для соответствующего актера в таблице actor_apps.

Код триггера представлен в [листинге 7](#).

Листинг 7. Код триггера add_app

```

1 CREATE TRIGGER delete_app
2 AFTER DELETE ON application
3 FOR EACH ROW
4 BEGIN
5     UPDATE actor_apps
6     SET num_apps = num_apps - 1
7     WHERE id_actor = OLD.id_actor;
8 END;

```

Для демонстрации работы данного триггера удалим заявку актеру с `id_actor = 1`. Для этого сначала нужно удалить все записи в других таблицах, которые ссылаются на первичный ключ таблицы `application`.

```

1 DELETE FROM doubles_audition WHERE id_application = 1;
2 DELETE FROM audition WHERE id_application = 1;
3 DELETE FROM first_stage WHERE id_application = 1;
4 DELETE FROM application WHERE id_application = 1;

```

Результат работы триггера представлен на [рисунке 7](#).

```

MySQL localhost:33060+ ssl casting SQL > select * from actor_apps where id_actor = 1;
+-----+-----+-----+-----+-----+
| id_actor | surname | name   | patronymic | num_apps |
+-----+-----+-----+-----+-----+
| 1       | Богдан  | Арсений | Вячеславович | 9       |
+-----+-----+-----+-----+-----+
1 row in set (0.0010 sec)

MySQL localhost:33060+ ssl casting SQL > DELETE FROM audition WHERE id_application = 1;
Query OK, 1 row affected (0.0109 sec)

MySQL localhost:33060+ ssl casting SQL > DELETE FROM first_stage WHERE id_application = 1;
Query OK, 2 rows affected (0.0040 sec)

MySQL localhost:33060+ ssl casting SQL > DELETE FROM application WHERE id_application = 1;
ERROR: 1451: Cannot delete or update a parent row: a foreign key constraint fails (`casting`.`doubles_aud
application`)

MySQL localhost:33060+ ssl casting SQL > DELETE FROM doubles_audition WHERE id_application = 1;
Query OK, 1 row affected (0.0078 sec)

MySQL localhost:33060+ ssl casting SQL > DELETE FROM application WHERE id_application = 1;
Query OK, 1 row affected (0.0076 sec)

MySQL localhost:33060+ ssl casting SQL > select * from actor_apps where id_actor = 1;
+-----+-----+-----+-----+-----+
| id_actor | surname | name   | patronymic | num_apps |
+-----+-----+-----+-----+-----+
| 1       | Богдан  | Арсений | Вячеславович | 8       |
+-----+-----+-----+-----+-----+
1 row in set (0.0011 sec)

```

Рис. 7. Результат работы триггера delete_app

3. Триггер add_actor

- Событие: Добавление нового актера в таблицу `actor`.
- Действие: Добавление данных нового актера в таблицу `actor_apps`.

- Принцип работы: при добавлении нового актера в таблицу actor срабатывает триггер, добавляющий данные нового актера в таблицу actor_apps.

Код триггера представлен в [листинге 8](#).

Листинг 8. Код триггера add_actor

```

1 CREATE TRIGGER add_actor
2 AFTER INSERT ON actor
3 FOR EACH ROW
4 BEGIN
5     INSERT INTO actor_apps (id_actor, surname, name, patronymic, num_apps)
6     VALUES (NEW.id_actor, NEW.surname, NEW.name, NEW.patronymic, 0);
7 END;
```

Для демонстрации работы данного триггера добавим нового актера в таблицу actor.

```

1 INSERT INTO actor(surname, name, patronymic, date_of_birth)
2 VALUES ('Соловьев', 'Михаил', 'Владимирович', '2000-02-13');
```

Результат работы триггера представлен на [рисунке 8](#).

```

MySQL localhost:33060+ ssl casting SQL > select * from actor_apps order by id_actor desc limit 5;
+-----+-----+-----+-----+-----+
| id_actor | surname | name | patronymic | num_apps |
+-----+-----+-----+-----+-----+
| 10000 | Крылова | Вера | Никитична | 6 |
| 9999 | Потапова | Елизавета | Макаровна | 5 |
| 9998 | Николаев | Тимофей | Алексеевич | 5 |
| 9997 | Прохорова | Мария | Руслановна | 6 |
| 9996 | Мартынов | Фёдор | Кириллович | 6 |
+-----+-----+-----+-----+-----+
5 rows in set (0.0012 sec)

MySQL localhost:33060+ ssl casting SQL > INSERT INTO actor(surname, name, patronymic, date_of_birth)
-> VALUES ('Соловьев', 'Михаил', 'Владимирович', '2000-02-13');
Query OK, 1 row affected (0.0109 sec)

MySQL localhost:33060+ ssl casting SQL > select * from actor_apps order by id_actor desc limit 5;
+-----+-----+-----+-----+-----+
| id_actor | surname | name | patronymic | num_apps |
+-----+-----+-----+-----+-----+
| 10002 | Соловьев | Михаил | Владимирович | 0 |
| 10000 | Крылова | Вера | Никитична | 6 |
| 9999 | Потапова | Елизавета | Макаровна | 5 |
| 9998 | Николаев | Тимофей | Алексеевич | 5 |
| 9997 | Прохорова | Мария | Руслановна | 6 |
+-----+-----+-----+-----+-----+
5 rows in set (0.0012 sec)
```

Рис. 8. Результат работы триггера add_actor

4. Триггер delete_actor

- Событие: Удаление актера из таблицы actor.
- Действие: Удаление данных актера из таблицы actor_apps.
- Принцип работы: при удалении актера из таблицы actor срабатывает триггер, удаляющий данные актера из таблицы actor_apps.

Код триггера представлен в [листинге 9](#).

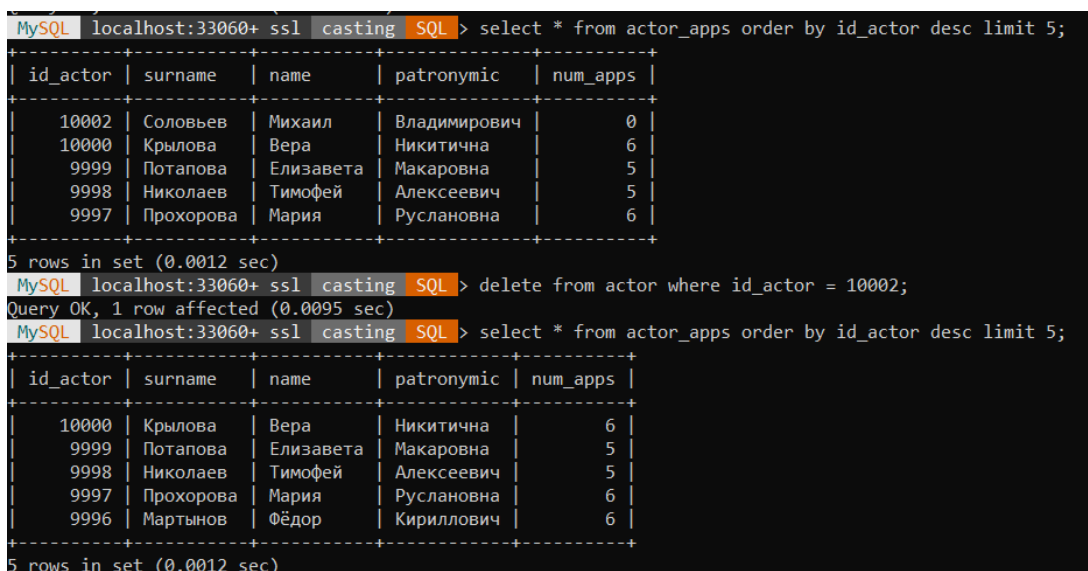
Листинг 9. Код триггера delete_actor

```
1 CREATE TRIGGER delete_actor
2 AFTER DELETE ON actor
3 FOR EACH ROW
4 BEGIN
5     DELETE FROM actor_apps
6     WHERE id_actor = OLD.id_actor;
7 END;
```

Для демонстрации работы данного триггера удалим актера из таблицы actor.

```
1 DELETE FROM actor WHERE id_actor = 10002;
```

Результат работы триггера представлен на [рисунке 9](#).



The screenshot shows a MySQL terminal session. The first query is a SELECT statement that returns 5 rows from the actor_apps table, ordered by id_actor in descending order. The second query is a DELETE statement that removes the row from the actor table where id_actor is 10002. The third query is another SELECT statement that returns 5 rows from the actor_apps table, ordered by id_actor in descending order. The result shows that the row with id_actor 10002 has been removed from the actor table, and the actor_apps table now contains 5 rows.

id_actor	surname	name	patronymic	num_apps
10002	Соловьев	Михаил	Владимирович	0
10000	Крылова	Вера	Никитична	6
9999	Потапова	Елизавета	Макаровна	5
9998	Николаев	Тимофей	Алексеевич	5
9997	Прохорова	Мария	Руслановна	6

5 rows in set (0.0012 sec)

```
MySQL localhost:33060+ ssl casting SQL > delete from actor where id_actor = 10002;
```

Query OK, 1 row affected (0.0095 sec)

```
MySQL localhost:33060+ ssl casting SQL > select * from actor_apps order by id_actor desc limit 5;
```

id_actor	surname	name	patronymic	num_apps
10000	Крылова	Вера	Никитична	6
9999	Потапова	Елизавета	Макаровна	5
9998	Николаев	Тимофей	Алексеевич	5
9997	Прохорова	Мария	Руслановна	6
9996	Мартынов	Фёдор	Кириллович	6

5 rows in set (0.0012 sec)

Рис. 9. Результат работы триггера delete_actor

5. Триггер update_actor

- Событие: Обновление данных актера в таблице actor.
- Действие: Обновление данных актера в таблице actor_apps.
- Принцип работы: при изменении данных актера в таблице actor срабатывает триггер, обновляющий данные актера в таблице actor_apps.

Код триггера представлен в [листинге 10](#).

Листинг 10. Код триггера update_actor

```

1 CREATE TRIGGER update_actor
2 AFTER UPDATE ON actor
3 FOR EACH ROW
4 BEGIN
5     UPDATE actor_apps
6     SET surname = NEW.surname ,
7         name = NEW.name ,
8         patronymic = NEW.patronymic
9     WHERE id_actor = NEW.id_actor ;
10 END;

```

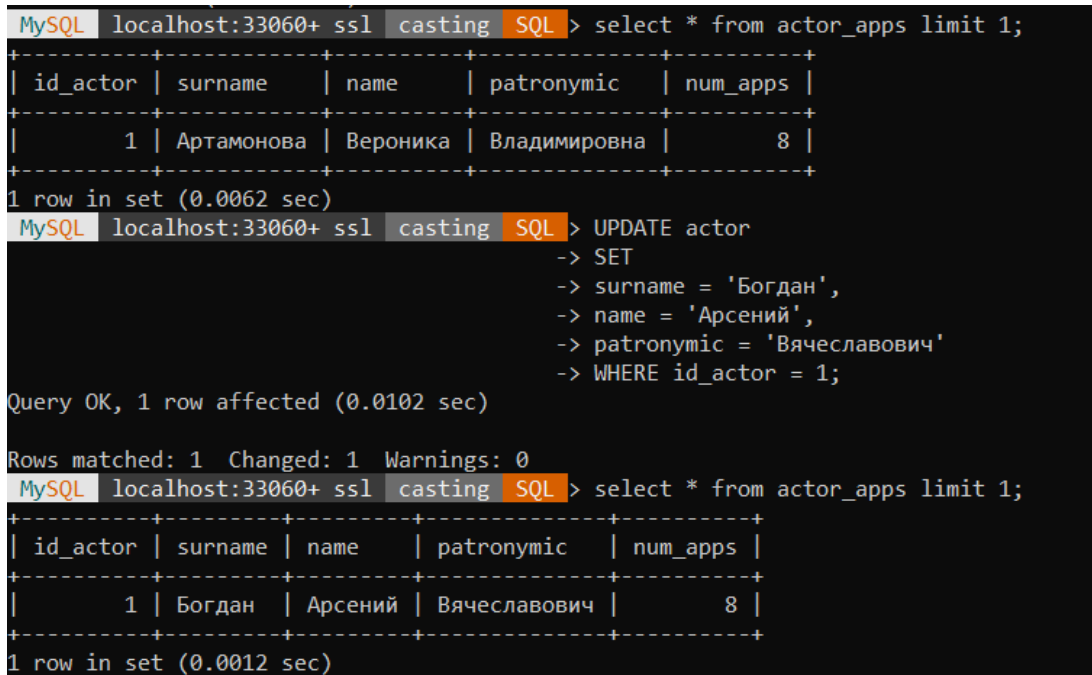
Для демонстрации работы данного триггера изменим данные актера в таблице actor.

```

1 UPDATE actor
2 SET
3     surname = 'Богдан',
4     name = 'Арсений',
5     patronymic = 'Вячеславович'
6 WHERE id_actor = 1;

```

Результат работы триггера представлен на рисунке 10.



```

MySQL localhost:33060+ ssl casting SQL > select * from actor_apps limit 1;
+-----+-----+-----+-----+-----+
| id_actor | surname | name | patronymic | num_apps |
+-----+-----+-----+-----+-----+
| 1 | Артамонова | Вероника | Владимировна | 8 |
+-----+-----+-----+-----+-----+
1 row in set (0.0062 sec)

MySQL localhost:33060+ ssl casting SQL > UPDATE actor
-> SET
-> surname = 'Богдан',
-> name = 'Арсений',
-> patronymic = 'Вячеславович'
-> WHERE id_actor = 1;

Query OK, 1 row affected (0.0102 sec)

Rows matched: 1 Changed: 1 Warnings: 0

MySQL localhost:33060+ ssl casting SQL > select * from actor_apps limit 1;
+-----+-----+-----+-----+-----+
| id_actor | surname | name | patronymic | num_apps |
+-----+-----+-----+-----+-----+
| 1 | Богдан | Арсений | Вячеславович | 8 |
+-----+-----+-----+-----+-----+
1 row in set (0.0012 sec)

```

Рис. 10. Результат работы триггера update_actor

2.3 Разграничение прав доступа

Задача: Создать двух пользователей. Первый должен уметь читать созданное ранее представление, а второй – редактировать таблицы, участвующие в нем.

В результате реализации задачи было создано два пользователя - **reader** и **editor**. Первому пользователю предоставляются права только на чтение, второму – на просмотр представления и редактирование таблиц представления director, film, role, application, actor.

Код создания пользователей и предоставление им прав представлен в [листинге 11](#).

Листинг 11. Код создания пользователей и назначения им прав

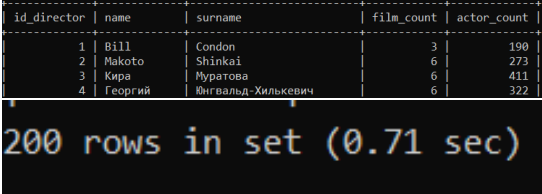
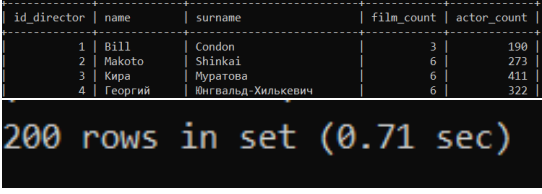

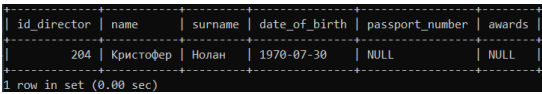


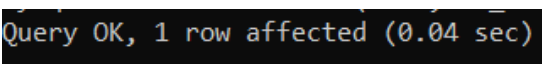
```
1 CREATE USER IF NOT EXISTS 'reader'@'localhost' IDENTIFIED BY '1111';
2 CREATE USER IF NOT EXISTS 'editor'@'localhost' IDENTIFIED BY '0000';
3
4 GRANT SELECT ON casting.director_view TO 'reader'@'localhost';
5
6 GRANT SELECT ON casting.director_view TO 'editor'@'localhost';
7
8 GRANT SELECT, INSERT, DELETE, UPDATE ON casting.film TO 'editor'@'localhost'
9   ⇨';
10 GRANT SELECT, INSERT, DELETE, UPDATE ON casting.role TO 'editor'@'localhost'
11   ⇨';
12 GRANT SELECT, INSERT, DELETE, UPDATE ON casting.application TO 'editor'@'
13   ⇨localhost';
14 GRANT SELECT, INSERT, DELETE, UPDATE ON casting.actor TO 'editor'@'
15   ⇨localhost';
```

Чтобы переключиться на другого пользователя используем команду в консоли:

```
1 mysql -u editor -p -h localhost
```

В [таблице 1](#) представлено сравнение реакций на различные действия пользователей reader и editor.

Таблица 1. Сравнение реакций на различные действия

№	editor	reader
1	<p>Просмотр таблицы director_view</p> <p>SELECT * from director_view;</p>	
		
2	<p>Добавление в таблицу director</p> <p>INSERT INTO director(name, surname, date_of_birth) VALUES ('Кристофер', 'Нолан', 1970-07-30);</p>	
		<p>ERROR 1142 (42000): INSERT command denied to user 'reader'@'localhost' for table 'director'</p>
3	<p>Результат добавления в таблицу director</p> <p>SELECT * FROM director ORDER BY id_director DESC LIMIT 1;</p>	
		<p>ERROR 1142 (42000): SELECT command denied to user 'reader'@'localhost' for table 'director'</p>
4	<p>Удаление из таблицы director</p> <p>DELETE FROM director WHERE id_director = 204;</p>	
		<p>ERROR 1142 (42000): DELETE command denied to user 'reader'@'localhost' for table 'director'</p>
5	<p>Результат удаления из таблицы director</p> <p>SELECT * FROM director WHERE id_director = 204;</p>	
		<p>ERROR 1142 (42000): SELECT command denied to user 'reader'@'localhost' for table 'director'</p>
6	<p>Добавление записи в таблицу film</p> <p>INSERT INTO film(name, id_director, id_casting_director) VALUES ('Социальная сеть', 1, 1);</p>	
		<p>ERROR 1142 (42000): INSERT command denied to user 'reader'@'localhost' for table 'film'</p>

№	editor	reader
7	Результат добавления записи в таблицу film SELECT * FROM film ORDER BY id_film DESC LIMIT 1;	
	<pre> +-----+-----+-----+-----+ id_film name id_director id_casting_director +-----+-----+-----+-----+ 1001 Социальная сеть 1 1 +-----+-----+-----+-----+ 1 row in set (0.01 sec) </pre>	ERROR 1142 (42000): SELECT command denied to user 'reader'@'localhost' for table 'film'
8	Удаление записи из таблицы application DELETE FROM application WHERE id_application = 59995;	
	<pre> Query OK, 1 row affected (0.04 sec) </pre>	ERROR 1142 (42000): DELETE command denied to user 'reader'@'localhost' for table 'application'
9	Результат удаления записи из таблицы application SELECT * FROM application WHERE id_application = 59995;	
	<pre> Empty set (0.02 sec) </pre>	ERROR 1142 (42000): SELECT command denied to user 'reader'@'localhost' for table 'application'
10	Удаление таблицы actor DROP table actor;	
	ERROR 1142 (42000): DROP command denied to user 'editor'@'localhost' for table 'actor'	ERROR 1142 (42000): DROP command denied to user 'reader'@'localhost' for table 'actor'
11	Добавление записи в таблицу genres INSERT INTO genres(name) VALUES ('Артхаус');	
	ERROR 1142 (42000): INSERT command denied to user 'editor'@'localhost' for table 'genres'	ERROR 1142 (42000): INSERT command denied to user 'reader'@'localhost' for table 'genres'
12	Изменение таблицы role UPDATE role SET name = 'Шерлок Холмс' WHERE id_role = 1;	
	<pre> Query OK, 1 row affected (0.01 sec) </pre>	ERROR 1142 (42000): UPDATE command denied to user 'reader'@'localhost' for table 'role'
13	Результат изменения записи в таблице role SELECT * FROM role WHERE id_role = 1;	
	<pre> +-----+-----+-----+-----+ id_role name id_role_type id_film +-----+-----+-----+-----+ 1 Шерлок Холмс 3 1 +-----+-----+-----+-----+ 1 row in set (0.01 sec) </pre>	ERROR 1142 (42000): SELECT command denied to user 'reader'@'localhost' for table 'role'

2.4 Функции и процедуры

2.4.1 Функция

Задача: Реализовать функцию, принимающую в качестве аргументов фамилию, имя и отчество и возвращающую строку формата "Фамилия.И.О."

Код создания функции представлен в [листинге 12](#).

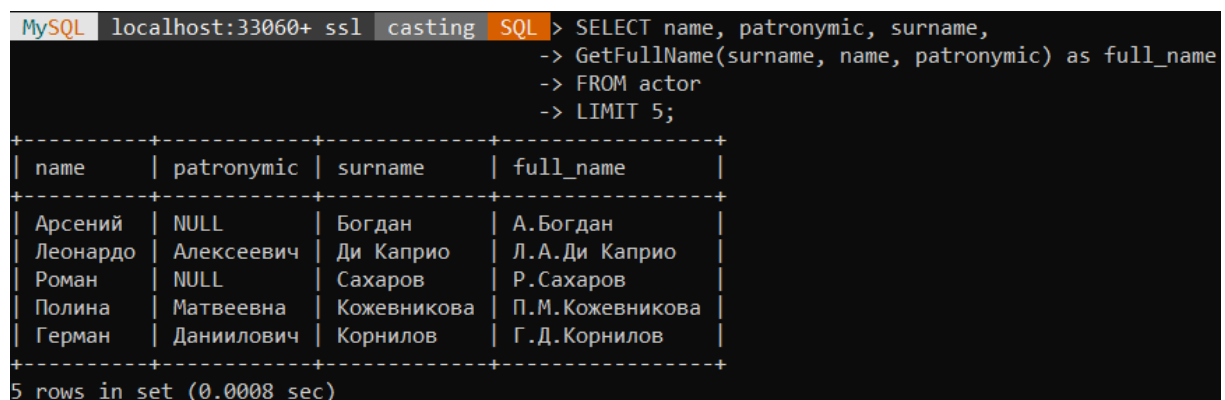
Листинг 12. Код создания функции GetFullName

```
1 DELIMITER //
2 CREATE FUNCTION GetFullName (
3     surname VARCHAR(30) ,
4     name VARCHAR(20) ,
5     patronymic VARCHAR(20)
6 ) RETURNS VARCHAR(35)
7 DETERMINISTIC
8 BEGIN
9     RETURN CONCAT(
10     LEFT(name, 1) , '. ' ,
11     IF(patronymic IS NOT NULL, CONCAT(LEFT(patronymic, 1) , '. ' ) , '' ) ,
12     surname
13 );
14 END //
15
16 DELIMITER ;
```

Пример использования функции:

```
1 SELECT surname, name, patronymic ,
2     GetFullName(surname, name, patronymic) as full_name
3 FROM actor
4 LIMIT 5;
```

Результат выполнения запроса с использованием функции представлен на [рисунке 11](#).



The screenshot shows a MySQL terminal window with the following command and output:

```
MySQL localhost:33060+ ssl casting SQL > SELECT name, patronymic, surname,
-> GetFullName(surname, name, patronymic) as full_name
-> FROM actor
-> LIMIT 5;
```

name	patronymic	surname	full_name
Арсений	NULL	Богдан	А.Богдан
Леонардо	Алексеевич	Ди Каприо	Л.А.Ди Каприо
Роман	NULL	Сахаров	Р.Сахаров
Полина	Матвеевна	Кожевникова	П.М.Кожевникова
Герман	Данилович	Корнилов	Г.Д.Корнилов

5 rows in set (0.0008 sec)

Рис. 11. Результат выполнения запроса с функцией

Если же вызвать функцию и передать попробовать ей атрибуты, которых нет в таблице, то получим ошибку. Пример такого ошибочного использования представлен ниже, а результат его выполнения на [рисунке 12](#).

```
1 SELECT GetFullName(surname, name, patronymic) as full_name
2 FROM application
3 LIMIT 5;
```

```
MySQL localhost:33060+ ssl casting SQL > SELECT GetFullName(surname, name, patronymic)
-> FROM application
-> LIMIT 5;
ERROR: 1054: Unknown column 'surname' in 'field list'
MySQL localhost:33060+ ssl casting SQL >
```

Рис. 12. Результат выполнения запроса

2.4.2 Процедура

Задача: Реализовать процедуру, добавляющую новую запись в таблицу application и принимающую в качестве аргументов фамилию, имя, отчество, дату рождения, номер паспорта, образование и опыт работы актера, а также имя, фамилия, дату рождения, номер паспорта и награды режиссера, id кастинг-директора и id роли. При вызове процедуры проверять, существует ли запись в таблице astor с такими же фамилией, именем и отчеством, и, если она существует, при необходимости обновить значения полей записи.

Код создания функции представлен в [листинге 13](#).

Листинг 13. Код создания процедуры new_application

```
1 DELIMITER //
2
3 CREATE PROCEDURE new_application(
4     IN ac_surname varchar(30),
5     IN ac_name varchar(20),
6     IN ac_patronymic varchar(20),
7     IN ac_date_of_birth date,
8     IN ac_passport_number varchar(15),
9     IN ac_education varchar(100),
10    IN ac_work_experience varchar(100),
11    IN d_surname varchar(30),
12    IN d_name varchar(20),
13    IN d_date_of_birth date,
14    IN d_passport_number varchar(15),
15    IN d_awards varchar(100),
16    IN id_cas_dir int unsigned,
17    IN id_input_role int unsigned
18 )
```

```

19 BEGIN
20     DECLARE act_id INT DEFAULT NULL;
21     DECLARE dir_id INT DEFAULT NULL;
22
23     SELECT id_actor INTO act_id
24         FROM actor
25         WHERE surname = ac_surname COLLATE utf8mb4_unicode_ci
26         AND name = ac_name COLLATE utf8mb4_unicode_ci
27         AND patronymic = ac_patronymic COLLATE utf8mb4_unicode_ci
28         AND date_of_birth = ac_date_of_birth;
29
30     IF act_id IS NULL THEN
31         INSERT INTO actor(surname, name, patronymic, date_of_birth,
32             passport_number, education, work_experience)
33         VALUES(ac_surname, ac_name, ac_patronymic, ac_date_of_birth,
34             ac_passport_number, ac_education, ac_work_experience);
35         SET act_id = LAST_INSERT_ID();
36
37     ELSEIF EXISTS (SELECT 1 FROM actor WHERE id_actor = act_id
38         AND (passport_number <> ac_passport_number
39         OR education <> ac_education
40         OR work_experience <> ac_work_experience)
41     ) THEN
42         UPDATE actor
43         SET passport_number = ac_passport_number, education = ac_education,
44             work_experience = ac_work_experience WHERE id_actor = act_id;
45     END IF;
46
47     SELECT id_director INTO dir_id
48         FROM director
49         WHERE surname = d_surname COLLATE utf8mb4_unicode_ci
50         AND name = d_name COLLATE utf8mb4_unicode_ci
51         AND date_of_birth = d_date_of_birth;
52
53     IF dir_id IS NULL THEN
54         INSERT INTO director(surname, name, date_of_birth, passport_number,
55             ⇨ awards)
56         VALUES(d_surname, d_name, d_date_of_birth, d_passport_number,
57             ⇨ d_awards);
58         SET dir_id = LAST_INSERT_ID();
59     ELSEIF EXISTS (SELECT 1 FROM director WHERE id_director = dir_id
60         AND (passport_number <> d_passport_number
61         OR awards <> d_awards)
62     ) THEN

```

```

61      UPDATE director
62      SET passport_number = d_passport_number, awards = d_awards
63      WHERE id_director = dir_id;
64  END IF;
65
66  — Вставка в таблицу application
67  INSERT INTO application(filmography, photos, id_actor, id_role,
68  ↪ id_casting_director, id_director)
69  VALUES(NULL, NULL, act_id, id_input_role, id_cas_dir, dir_id);
70 END //
71
72 DELIMITER ;

```

Продemonстрируем результаты работы процедуры для следующих случаев:

1. Актер и режиссер уже существуют в таблицах actor и director, но некоторые их данные **НЕ совпадают** с входными значениями аргументов процедуры;
2. Актер и режиссер уже существуют в таблицах actor и director, и все данные **совпадают** с входными значениями аргументов вызываемой процедуры;
3. Актера и режиссера **не существует** в таблицах actor и director до вызова процедуры.

2.4.2.1 Первый пример

Для демонстрации **первого** примера, сначала посмотрим, какие заявки имеются у актера с id_actor = 2. Для этого выполним следующий запрос:

```

1 SELECT a.id_actor, a.surname, a.name, app.id_application,
2       r.id_role, dir.id_director
3 FROM actor a
4 LEFT JOIN application app on a.id_actor = app.id_actor
5 JOIN director dir on app.id_director = dir.id_director
6 JOIN role r on app.id_role = r.id_role
7 WHERE a.id_actor = 2;

```

Также выполним запросы для просмотра всей информации об актере и режиссере:

```

1 SELECT * from actor where id_actor = 2;
2
3 SELECT * from director where id_director = 54;

```

Результаты выполнения запросов представлены на [рисунках 13-15](#).

id_actor	surname	name	id_application	id_role	id_director
2	Ди Каприо	Леонардо	9	8553	188
2	Ди Каприо	Леонардо	10	2429	29
2	Ди Каприо	Леонардо	11	1421	11
2	Ди Каприо	Леонардо	12	924	63
2	Ди Каприо	Леонардо	13	11819	116
2	Ди Каприо	Леонардо	14	5288	195

6 rows in set (0.01 sec)

Рис. 13. Данные о заявках актера

```
mysql> SELECT * from actor where id_actor = 2;
```

id_actor	surname	name	patronymic	date_of_birth	passport_number	education	work_experience
2	Ди Каприо	Леонардо	NULL	2004-06-21	1111 11111	Высшее образование СПбПУ	Нет опыта

1 row in set (0.00 sec)

Рис. 14. Данные актера

```
mysql> SELECT * from director where id_director = 54;
```

id_director	name	surname	date_of_birth	passport_number	awards
54	Christopher	Nolan	1978-11-03	5942 113007	Номинант на премию "Сезар"

1 row in set (0.00 sec)

Рис. 15. Данные режиссера

Теперь вызовем процедуру:

```
1 CALL new_application(
2     'Ди Каприо', 'Леонардо', 'Алексеевич', '2004-06-21',
3     '1234567890', 'СПбПУ + Oxford',
4     'Снялся в фильме Титаник',
5     'Nolan', 'Christopher', '1978-11-03', '0987654321',
6     'Оскар (лучший актер)',
7     1, 1
8 );
```

Теперь посмотрим на результат, выполнив запросы, уже показанные выше. Результат представлен на [рисунках 16-18](#).

id_actor	surname	name	id_application	id_role	id_director
2	Ди Каприо	Леонардо	9	8553	188
2	Ди Каприо	Леонардо	10	2429	29
2	Ди Каприо	Леонардо	11	1421	11
2	Ди Каприо	Леонардо	12	924	63
2	Ди Каприо	Леонардо	13	11819	116
2	Ди Каприо	Леонардо	14	5288	195
2	Ди Каприо	Леонардо	59997	1	54

7 rows in set (0.0011 sec)

Рис. 16. Данные о заявках актера

id_actor	surname	name	patronymic	date_of_birth	passport_number	education	work_experience
2	Ди Каприо	Леонардо	Алексеевич	2004-06-21	1234567890	СПбПУ + Oxford	Снялся в фильме Титаник

1 row in set (0.0011 sec)

Рис. 17. Данные актера

id_director	name	surname	date_of_birth	passport_number	awards
54	Christopher	Nolan	1978-11-03	0987654321	Оскар (лучший актер)

1 row in set (0.0012 sec)

Рис. 18. Данные режиссера

2.4.2.2 Второй пример

Для демонстрации **второго** случая изменим в вызове процедуры id_role=2 и id_casting_director=2:

```

1 CALL new_application(
2   'Ди Каприо', 'Леонардо', 'Алексеевич', '2004-06-21', '1234567890',
3   'СПбПУ + Oxford',
4   'Снялся в фильме Титаник',
5   'Nolan', 'Christopher',
6   '1978-11-03',
7   '0987654321',
8   'Оскар (лучший актер)',
9   1, 1
10  );

```

Результат выполнения процедуры представлен на [рисунке 19](#).

id_actor	surname	name	id_application	id_role	id_director
2	Ди Каприо	Леонардо	9	8553	188
2	Ди Каприо	Леонардо	10	2429	29
2	Ди Каприо	Леонардо	11	1421	11
2	Ди Каприо	Леонардо	12	924	63
2	Ди Каприо	Леонардо	13	11819	116
2	Ди Каприо	Леонардо	14	5288	195
2	Ди Каприо	Леонардо	59997	1	54
2	Ди Каприо	Леонардо	59998	2	54

8 rows in set (0.0022 sec)

Рис. 19. Данные о заявках актера

2.4.2.3 Третий пример

Для демонстрации **третьего** случая вызовем процедуру, указав новые данные актера и режиссера:

```

1 CALL new_application(
2     'Дауни-младший', 'Роберт', '', '1965-04-04', '1234567890',
3     'Cambridge University',
4     'Снялся в Шерлок Холмсе',
5     'Квентин', 'Тарантино', '1978-11-03', '0987654321',
6     'Золотой глобус', 10, 10
7 );

```

Для демонстрации результатов выполнения процедуры посмотрим последнюю запись из таблиц actor и director, а также посмотрим дополнительную информацию о заявках нового актера.(рис. 20-22).

```
MySQL localhost:33060+ ssl casting SQL > select * from actor order by id_actor desc limit 1;
```

id_actor	surname	name	patronymic	date_of_birth	passport_number	education	work_experience
10004	Дауни-младший	Роберт		1965-04-04	1234567890	Cambridge University	Снялся в Шерлок Холмсе

1 row in set (0.0008 sec)

Рис. 20. Данные нового добавленного актера

```
MySQL localhost:33060+ ssl casting SQL > select * from director order by id_director desc limit 1;
```

id_director	name	surname	date_of_birth	passport_number	awards
203	Тарантино	Квентин	1978-11-03	0987654321	Золотой глобус

1 row in set (0.0007 sec)

Рис. 21. Данные нового добавленного режиссера

id_actor	surname	name	id_application	id_role	id_director
10004	Дауни-младший	Роберт	59999	10	203

1 row in set (0.0009 sec)

Рис. 22. Данные о заявках добавленного актера

2.5 Транзакции

Задача: Проверить уровень изоляции READ COMMITTED на наличие неповторяющихся и фантомных чтений.

READ COMMITTED — это уровень изоляции транзакций, при котором каждая транзакция видит только те изменения, которые уже были зафиксированы другими транзакциями. При таком уровне изоляции возможны две основные проблемы: неповторяющиеся и фантомные чтения.

Проверим возникновение фантомного чтения

Результат проведения транзакций представлен в [таблице 2](#).

Таблица 2. Проведение транзакций

№	Транзакция №1	Транзакция №2
1	Установка уровня изоляции на READ COMMITTED	
	SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;	SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
1	Начало транзакций	
	START TRANSACTION;	START TRANSACTION;
2	Читаем значения из таблицы role_type	
	<pre>SELECT * FROM role_type; mysql> select * from role_type; +-----+-----+ id_role_type name +-----+-----+ 1 Главная 2 Второстепенная 3 Эпизодическая +-----+-----+ 3 rows in set (0.00 sec)</pre>	

№	Транзакция №1	Транзакция №2
3	Добавляем новую запись в таблицу role_type и сохраняем результат	
		INSERT INTO role_type(name) VALUES('Камео');
3	Проверим добавление новой записи в таблицу role_type	
		<pre>mysql> select * from role_type; +-----+-----+ id_role_type name +-----+-----+ Главная Второстепенная Эпизодическая Камео +-----+-----+ 4 rows in set (0.00 sec)</pre>
3	Фиксируем внесенные изменения	
		COMMIT;
4	Читаем значения из таблицы role_type и получаем фантомное чтение	
	SELECT * FROM role_type; <pre>mysql> select * from role_type; +-----+-----+ id_role_type name +-----+-----+ Главная Второстепенная Эпизодическая Камео +-----+-----+ 4 rows in set (0.00 sec)</pre>	

Таблица иллюстрирует выполнение транзакций при уровне изоляции READ COMMITTED. Сначала устанавливается уровень изоляции, затем обе транзакции начинают выполнение. Транзакция №2 добавляет запись в таблицу role_type, после чего фиксирует изменения. Транзакция №1, выполняя повторное чтение данных, обнаруживает **фантомное чтение**, когда новая запись становится видимой, добавленная другой транзакцией, хотя в момент первоначального чтения эти данные не существовали.

Заключение

В ходе курсовой работы были выполнены 5 лабораторных работ.

1. Создано представление, в котором для каждого режиссера подсчитывается общее количество его фильмов и актеров, участвующих в его фильмах. Также было написано два запроса, добавляющие к представлению новую информацию: общее количество жанров и ролей в фильмах каждого режиссера и количество утвержденных заявок для фильмов каждого режиссера. Количество записей в представлении: 200.
2. Написаны 5 триггеров, поддерживающие целостность данных созданной таблицы, в которой для каждого актера подсчитывается число заявок. Количество записей в созданной таблице: 10002.
3. Созданы два пользователя: reader и editor. Первый имеет доступ только на просмотр представления, а второй также может редактировать таблицы, из которых оно состоит.
4. Написаны функция и процедура. Функция принимает в качестве аргументов фамилию, имя и отчество и возвращает строку формата "Фамилия И. О.". Процедура принимает в качестве аргументов фамилию, имя, отчество, дату рождения, номер паспорта, образование и опыт работы актера, а также имя, фамилия, дату рождения, номер паспорта и награды режиссера, id кастинг-директора и id роли. При вызове процедуры, если данные об актере и режиссере отсутствуют, должны создаваться соответствующие записи в таблицах actor и director. Если же данные об актере и режиссере есть, но с изменениями в некоторых полях, применяет новые изменения. После в таблице application создается новая запись с id соответствующих записей.
5. Проведена демонстрация возможности наличия фантомного чтения на уровне READ COMMITTED. Для этого были запущены две транзакции с разных пользователей, работающие одновременно с одной таблице role_type.

В процессе выполнения лабораторных работ было написано 32 запроса. Для наглядного иллюстрирования процесса работы и результатов каждого задания суммарно в отчёте представлено 22 рисунка, 13 листингов и 2 таблицы. Времени потрачено на выполнение всех заданий вместе с написанием отчета на языке разметки LaTeX: 30 часов.

Список литературы

- [1] Мана Такахаси, Сёко Адзума. Базы данных. - Москва: ДМК Пресс, 2015. - 240 с.
- [2] Силверман, Бен. MySQL. Библия пользователя. - М.: ДМК Пресс, 2019. - 928 с.
- [3] Бейли, Ларри Ульман. Изучаем MySQL. - 3-е издание. - СПб.: Питер, 2019. - 736 с.