МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ  
Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ   
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Информационных технологий

Кафедра Информационных систем и технологий

Специальность 1-98 01 03 «Программное обеспечение информационной безопасности мобильных систем»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА КУРСОВОГО ПРОЕКТА**

по дисциплине «Базы данных»

Тема «Реализация базы данных сети кинотеатров»

**Исполнитель**

студент 2 курса 7 группы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Лагун А. И.

подпись, дата

**Руководитель**

Доцент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Блинова Е. А.

должность, учен. степень, ученое звание подпись, дата

Допущен к защите \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

дата, подпись

Курсовой проект защищен с оценкой

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_ Блинова Е. А.

подпись дата инициалы и фамилия

**Содержание**

[Введение 5](#_Toc166740616)

[1. Постановка задачи и анализ требований 6](#_Toc166740617)

[1.1 Цели и задачи проекта 6](#_Toc166740618)

[1.2 Аналитический обзор аналогов 7](#_Toc166740619)

[1.2.1 Аналог «Skyline». 7](#_Toc166740623)

[1.2.2 Аналог «mooon». 9](#_Toc166740624)

[1.3 Разработка функциональных требований, определение вариантов использования 9](#_Toc166740625)

[1.4 Вывод по разделу 10](#_Toc166740626)

[2. Проектирование и разработка базы данных 11](#_Toc166740627)

[2.1 Диаграммы UML, взаимосвязь всех компонентов 11](#_Toc166740628)

[2.2 Роли и пользователи 12](#_Toc166740629)

[2.3 Привилегии. 12](#_Toc166740630)

[2.4 Вывод по разделу. 13](#_Toc166740631)

[3. Разработка необходимых объектов. 14](#_Toc166740632)

[3.1 Описание информационных объектов и ограничений целостности. 14](#_Toc166740633)

[3.1.1 Таблица INFO\_FILMS 14](#_Toc166740634)

[3.1.2 Таблица RATINGS 14](#_Toc166740635)

[3.1.3 Таблица ADMINISTRATORS 15](#_Toc166740636)

[3.1.4 Таблица CUSTOMERS 15](#_Toc166740637)

[3.1.5 Таблица THEATERS 15](#_Toc166740638)

[3.1.6 Таблица HALLS 15](#_Toc166740639)

[3.1.7 Таблица TIMETABLE 16](#_Toc166740640)

[3.1.8 Таблица ORDERS 16](#_Toc166740641)

[3.1.9 Таблица ORD\_DET 16](#_Toc166740642)

[3.1.10 Таблица TICKETS 16](#_Toc166740643)

[3.1.11 Таблица HISTORY 17](#_Toc166740644)

[3.2 Процедуры. 17](#_Toc166740645)

[3.3 Функции 18](#_Toc166740646)

[3.4 Триггеры 20](#_Toc166740647)

[3.5 Представление 21](#_Toc166740648)

[3.6 Вывод 21](#_Toc166740649)

[4. Описание процедур экспорта и импорта. 22](#_Toc166740650)

[4.1 Процедура экспорта данных 22](#_Toc166740651)

[4.2 Процедура импорта данных 23](#_Toc166740652)

[4.3 Вывод 24](#_Toc166740653)

[5. Тестирование производительности базы данных. 25](#_Toc166740654)

[5.1 Тестирование производительности по таблице TICKETS. 25](#_Toc166740655)

[5.2 Создание индекса 27](#_Toc166740656)

[5.3 Вывод. 28](#_Toc166740657)

[6. Описание технологии и ее применение в базе данных 29](#_Toc166740658)

[6.1 Технология Full Text Search 29](#_Toc166740659)

[6.2 Вывод 30](#_Toc166740660)

[7. Сценарий использования. 31](#_Toc166740661)

[7.1 Сценария использования для управляющего. 31](#_Toc166740662)

[7.2 Вывод 32](#_Toc166740663)

[Заключение 33](#_Toc166740664)

[Приложение А 35](#_Toc166740665)

[Приложение Б 39](#_Toc166740666)

[Приложение В 65](#_Toc166740667)

Введение

Целью данной работы являлась разработка реляционной базы данных для сети кинотеатров с определенным функционалом. База данных предназначена для обеспечения доступа клиентов к совершению покупок и управлению данными.

Написание базы данных для сети кинотеатров остается актуальным и востребованным в настоящее время из-за нескольких причин. Во-первых, такая база данных позволяет эффективно управлять информацией о фильмах, сеансах, билетах и покупателях, облегчая процессы поиска и продажи билетов. Во-вторых, разработка базы данных автоматизирует множество операций, таких как поиск билетов, учет проданных билетов, расчеты стоимости. Это повышает эффективность работы персонала и улучшает общий опыт клиентов. Кроме того, база данных может быть интегрирована с другими системами, что позволяет обеспечить единый и согласованный доступ к информации.

База данных — это организованная структура, предназначенная для хранения информации, систематизированная таким образом, чтобы эти материалы могли быть найдены и обработаны с помощью электронной вычислительной машины. Реляционная база данных — база данных, основанная на реляционной модели данных. В качестве СУБД для базы данных была выбрана PostgreSQL, благодаря ее высокой производительности и надежности.

Для реализации доступа к функционалу базы данных будет написан скрипт, использующий вызовы соответствующих функций. Скрипт будет обеспечивать взаимодействие с базой данных PostgreSQL, и его функциональность будет соответствовать требованиям, определенным заранее.

В основной части будут затронуты все аспекты разработки проекта и обоснованы некоторые технические приёмы, к которым приходилось прибегнуть, с целью реализации работы веб-сервера с базой данных.

Для повышения комфорта использования пользователями приложения в моем курсовом проекте используется технология применения Full Text Search-технологий СУБД в реляционной базе данных.

В пояснительной записке вы сможете найти краткую информацию о похожих продуктах, архитектуре, реализации проекта, руководстве пользователя.

# **1. Постановка задачи и анализ требований**

1.1 Цели и задачи проекта

Целью данного курсового проекта является создание базы данных для сети кинотеатров.

Задача проекта: создание структуры базы данных, включая таблицы для хранения информации о кинотеатрах, фильмах, сеансах, билетах и продажах. Реализация процедур и функций для управления данными, включая добавление, обновление, удаление и извлечение информации. Создание пользовательских ролей и привилегий для различных уровней доступа к данным (для покупателя, администратора, управляющего сетью).

Функционально должны быть выполнены следующие задачи:

* Возможность поиска билетов на киносеанс по различным параметрам, таким как название, стоимость, дата показа, жанр, наличие нужного количества доступных билетов, возможность сортировки фильмов по рейтингу. Реализовать механизм внесения изменений в список, таких как добавление, изменение и удаление сеансов;
* Определение трёх ролей: управляющий сетью кинотеатров, администратор, покупатель;
* Управляющий сетью кинотеатров может изменять список фильмов в прокате, регулировать цену билета на сеанс, добавлять, удалять администраторов;
* Администратор может добавлять данные о покупателях, рассчитывать итоговую стоимость набора билетов, отмечать количество проданных билетов, изменять количество доступных билетов, отмечать время и дату покупки;
* Покупатель может просматривать количество доступных билетов, их стоимость, покупать, возвращать билеты, ставить оценку на фильм.

Должны быть выполнены следующие требования:

* база данных должна быть спроектирована в СУБД PostgreSQL.
* доступ к данным должен осуществляться только через соответствующие процедуры;
* должен быть проведен импорт данных из XML файлов, экспорт данных в формат XML;
* необходимо протестировать производительность базы данных (на таблицах, содержащих не менее 100 000 строк) и внести изменения в структуру в случае необходимости.

Применить технологию базы данных согласно выбранной теме: подробно описать применяемые системные пакеты, утилиты или технологии; показать применение указанной технологии в базе данных.

Следует разработать три уровня доступа: управляющий, администратор, покупатель. Управляющий сетью кинотеатров может изменять список фильмов в прокате, регулировать цену билета на сеанс, добавлять, удалять администраторов. Администратор может добавлять данные о покупателях, рассчитывать итоговую стоимость набора билетов, отмечать количество проданных билетов, изменять количество доступных билетов, отмечать время и дату покупки. Покупатель может просматривать количество доступных билетов на фильм по различным параметрам, их стоимость, покупать, возвращать билеты, ставить оценку на фильм.

1.2 Аналитический обзор аналогов

Немаловажным этапом в разработке программного продукта является аналитический обзор прототипов и литературных источников.

На сегодняшний день можно встретить достаточно большое количество Web-решений, разработанных для сетей кинотеатров, но довольно малое количество десктопных программ, имеющих интуитивно понятный интерфейс. Были рассмотрены несколько программ-аналогов:

* «Skyline»;
* «mooon».



1.2.1 Аналог «Skyline».

«Skyline» [1] — веб-сайт, предоставляющий пользователям информацию о кинотеатрах, сеансах и позволяющий клиенту выполнять манипуляции с билетами. «Skyline» включает в себя ряд баз данных, которые хранят информацию о различных аспектах, эти данные можно логически разбить на блоки - таблицы. Вот некоторые из них:

Таблица "Кинотеатры": содержит информацию о кинотеатрах данной сети, такую как название, адрес, время работы, количество залов и фотографии.

Таблица "Залы": хранит информацию о залах, находящихся в различных кинотеатрах, их вместимости, лучших местах.

Таблица "Каталог фильмов": содержит информацию о всех фильмах в каталоге, включая их название, описание, рейтинг, жанр.

Таблица "Администраторы": содержит информацию об администраторах, которые управляют различными аспектами работы сети.

Таблица "Клиенты": содержит информацию о клиентах, включая их персональные данные, контактную информацию, историю покупок.

Таблица "Сеансы": содержит информацию о времени показа фильма, названии фильма, доступных билетах.

В сети кинотеатров данные хранятся в различных таблицах и в различных форматах, включая структурированные таблицы, текстовые файлы, аудио- и видеозаписи, фотографии и прочие форматы. Для доступа к базе данных используются специализированные программные интерфейсы и приложения, которые обеспечивают управление и обработку информации на различных этапах работы кинотеатров.

Интерфейс «Skyline» представлен на рисунке 1.1.

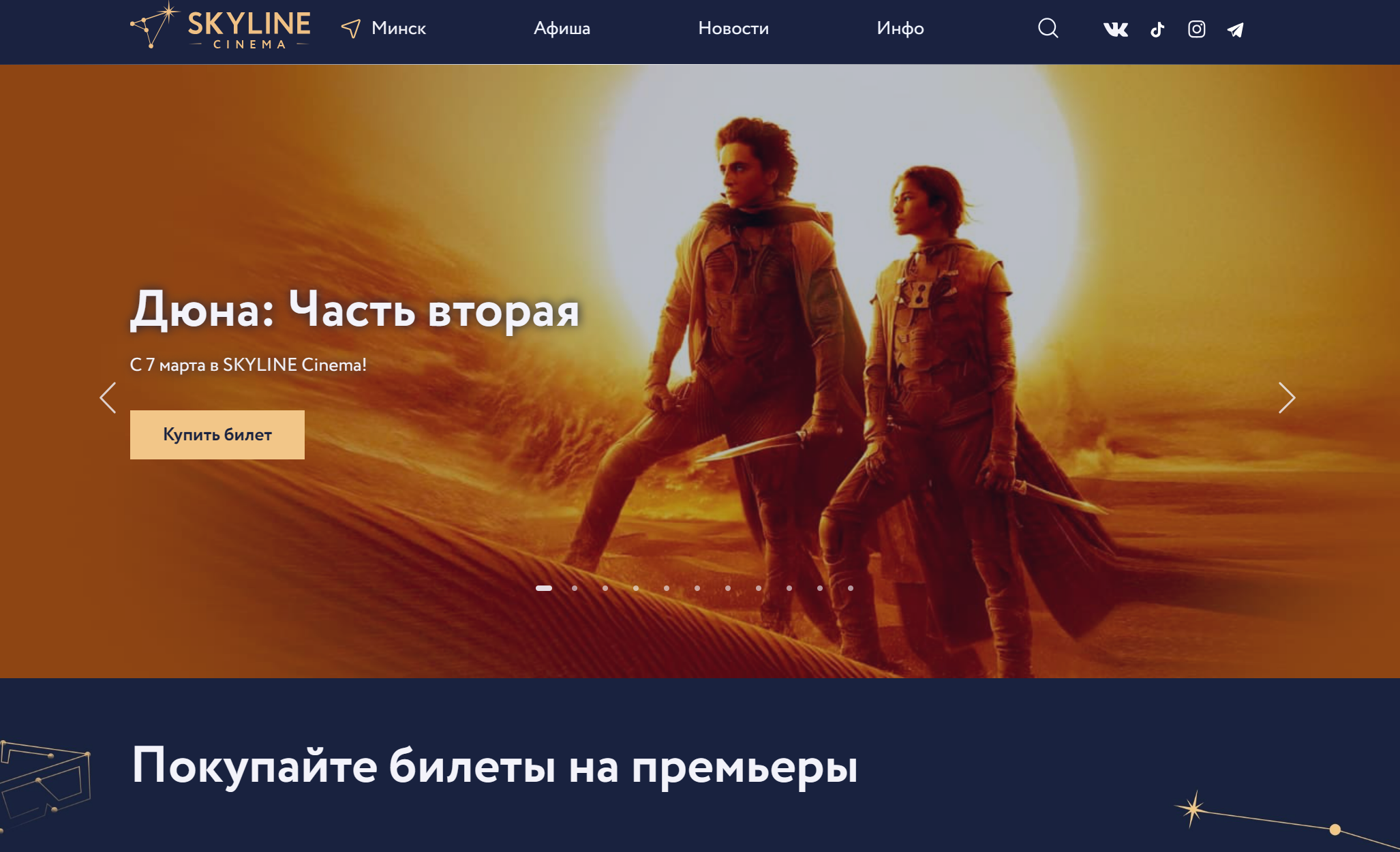


Рисунок 1.1 − Интерфейс «Skyline»

Каждый фильм на данным сайте содержит всю необходимую информацию, однако не содержит сведения о рейтинге. Соответственно фильтрация фильмов по рейтингу отсутствует.

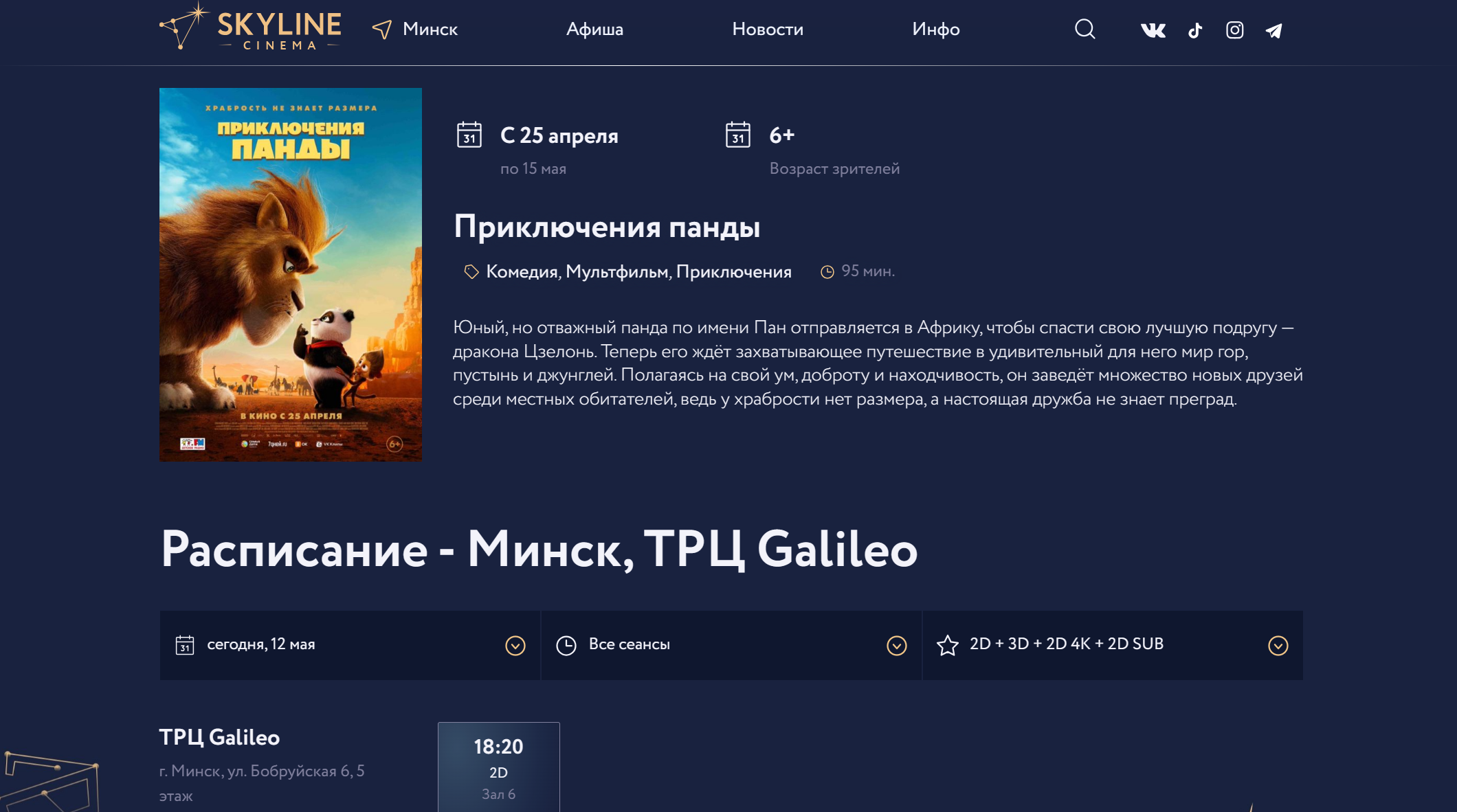


Рисунок 1.2 − Информация о фильме на сайте «Skyline»

Как и в случае любого подобного сайта, сеть кинотеатров также имеет некоторые недостатки. Один из таких недостатков может быть связан с отсутствием фильтрации фильмов по рейтингу, так же как и оценкой самого фильма. Некоторые пользователи могут столкнуться с проблемами при отображении медиа файлов. Например, возможны ошибки отображения видеороликов в каталоге.

1.2.2 Аналог «mooon».

Еще одним аналогом выбран веб-сервис «mooon» [2], который показан на рисунке 1.2. Продукт «mooon» — веб-сервис, позволяющий покупать билеты, просматривать информацию о кинотеатрах сети, сеансах.

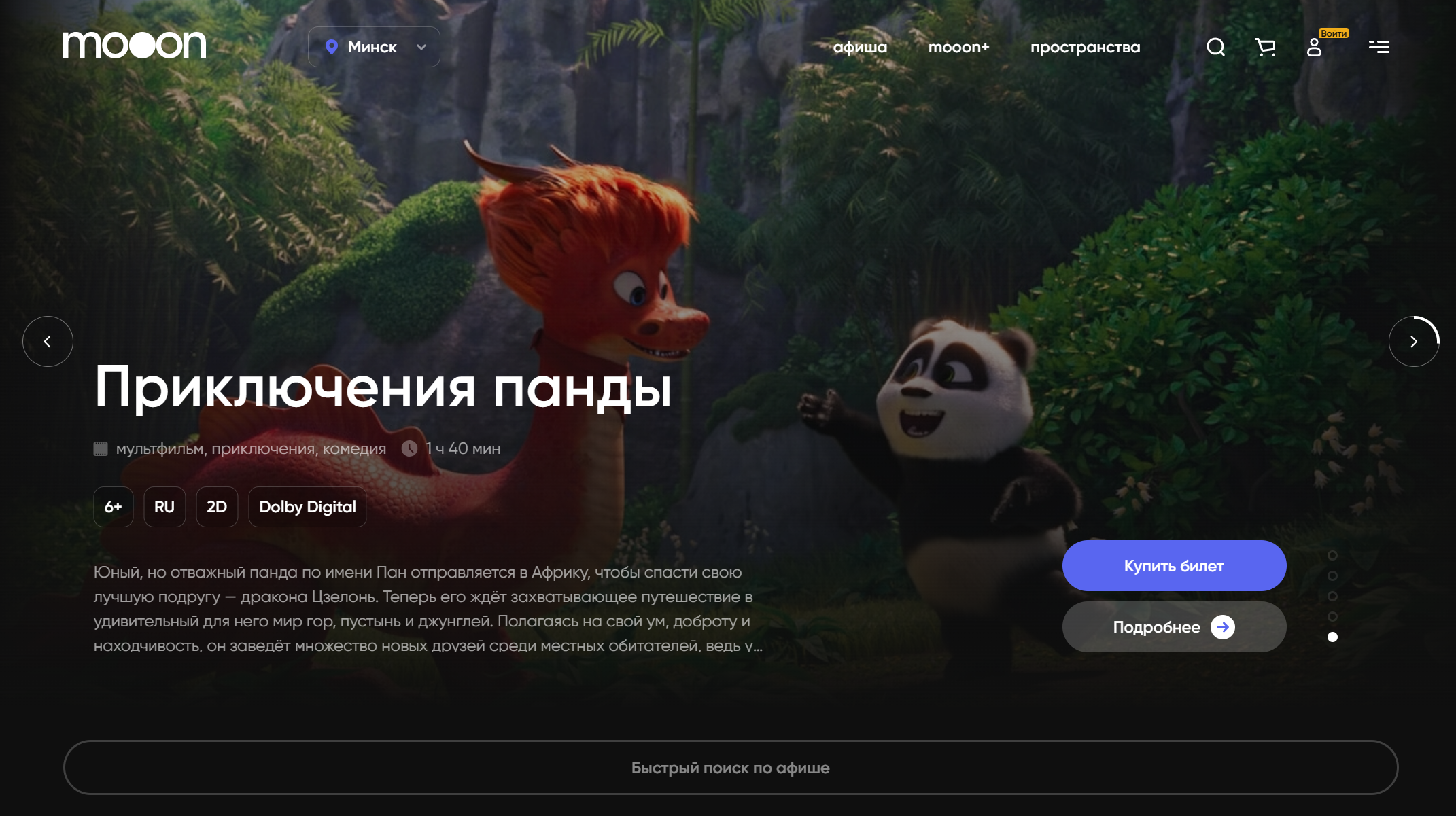


Рисунок 1.3 – Сайт «mooon»

Из минусов, можно отметить, также проблему отображения медиа файлов.

Плюсом данного аналога является предоставление обширной информации о кинопространстве и простота взаимодействия пользователя с сайтом для достижения нужной цели.

1.3 Разработка функциональных требований, определение вариантов использования

Функционально должны быть выполнены следующие задачи:

* Возможность поиска билетов на киносеанс по различным параметрам, таким как название, стоимость, дата показа, жанр, наличие нужного количества доступных билетов, возможность сортировки фильмов по рейтингу. Реализовать механизм внесения изменений в список, таких как добавление, изменение и удаление сеансов
* Определение трёх ролей: управляющий сетью кинотеатров, администратор, покупатель.
* Управляющий сетью кинотеатров может изменять список фильмов в прокате, регулировать цену билета на сеанс, добавлять, удалять администраторов.
* Администратор может добавлять данные о покупателях, рассчитывать итоговую стоимость набора билетов, отмечать количество проданных билетов, изменять количество доступных билетов, отмечать время и дату покупки.
* Покупатель может просматривать количество доступных билетов, их стоимость, покупать, возвращать билеты, ставить оценку на фильм.

В ходе изучения требований было выявлено, что база данных сети кинотеатров должна включать информацию о кинотеатрах, залах, фильмах, сеансах, билетах, заказах, покупателях, администраторах. Были определены следующие варианты использования:

* Хранение информации о фильмах, кинотеатрах, залах, сеансах, билетах, покупателях и администраторах.
* Просмотр информации о фильмах, сеансах и доступных на них билетах.
* Поиск нужных сеансов.
* Заказ и возврат билетов на сеанс.
* Отслеживание проданных билетов.
* Просмотр купленных пользователем билетов.
* Добавление покупателем оценки фильма.
* Добавление, обновление и удаление информации о фильмах, сеансах.
* Добавление и удаление администраторов из базы данных.

1.4 Вывод по разделу

В данном разделе были проведены исследования аналогичных решений, проанализированы их достоинства и недостатки, а также разработаны функциональные требования и была поставлена задача курсового проекта. Все, описанное в этом разделе, важно для создания высококачественного продукта.

# **2. Проектирование и разработка базы данных**

2.1 Диаграммы UML, взаимосвязь всех компонентов

В данной схеме представлены варианты взаимодействия с базой данных.

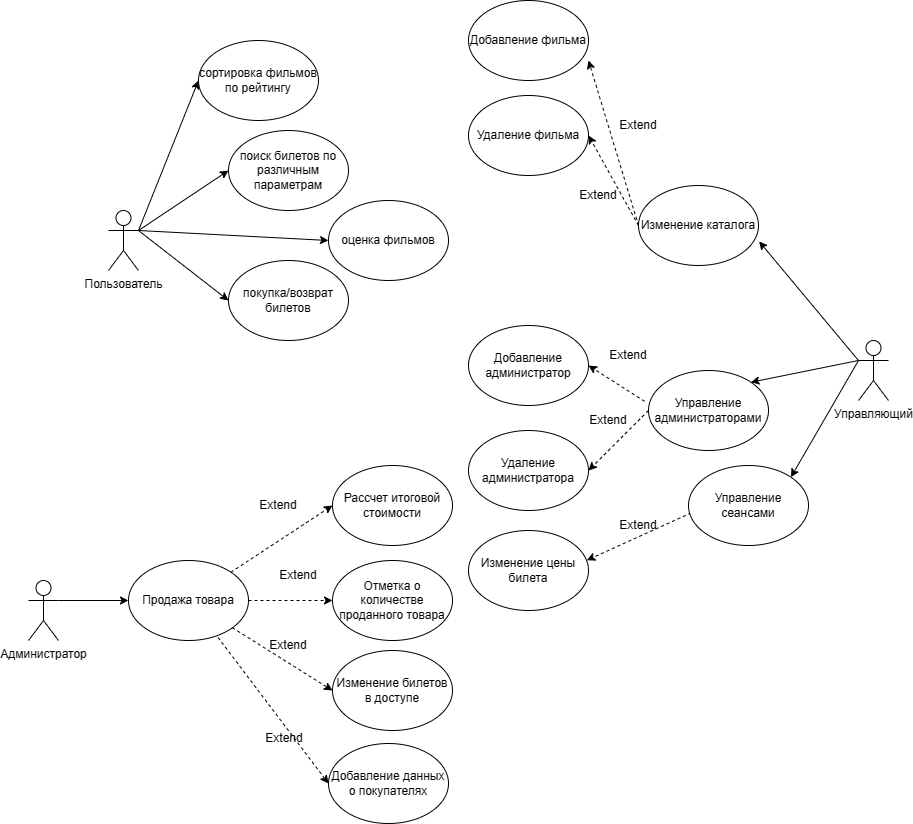


Рисунок 2.1 – Диаграмма использования

В возможности администратора входит:

* Расчет итоговой стоимости;
* Отметка о количестве проданных билетов;
* Изменение билетов в доступе;
* Добавление данных о покупателях.

В возможности пользователя входит:

* Поиск билетов по различным параметрам;
* Сортировка фильмов по рейтингу;
* Оценка фильма;
* Покупка/возврат билетов.

В возможности управляющего входит:

* Изменение фильмов в каталоге;
* Регулирование цены билета на сеанс;
* Добавлять/удалять администраторов.

2.2 Роли и пользователи

Роль — это поименованный набор привилегий. Роли нужны для упрощения процесса управления доступом и безопасности в базе данных.

Пользователь — это учетная запись, которая позволяет конкретному человеку или приложению получать доступ к базе данных. Присвоение пользователю ролей и профилей безопасности позволяет определить его права доступа и ограничения при работе с базой данных. Присвоение ролей и профилей пользователям позволяет управлять доступом к базе данных и обеспечить ее безопасность.

При выполнении команды CREATE USER в PostgreSQL создается пользователь базы данных, и автоматически создается роль с тем же названием. В PostgreSQL пользователь и роль совпадают по имени и используются взаимозаменяемо.

Пример создания роли и пользователя приведено на листинге 2.1.

|  |
| --- |
| create user manager with password 'qwerty' |

Листинг 2.1 – Создание роли «Управляющий»

Были разработаны следующие роли:

* Программист;
* Управляющий сетью;
* Администратор;
* Покупатель.

2.3 Привилегии.

Привилегии, выдаваемые пользователям, определяют, какие действия могут быть выполнены в отношении определенных объектов базы данных (например, таблиц, представлений и т.д.) и контролируют доступ пользователей к этим объектам. Таким образом, выдача привилегий ролям позволяет определить права доступа на уровне пользователей, что упрощает управление безопасностью и снижает риски нарушения безопасности базы данных.

Предоставление привилегий пользователям представлено на листинге 2.2.

grant execute on procedure add\_administrator to manager;

grant execute on procedure add\_movie\_to\_catalog to manager;

grant execute on procedure delete\_admin to manager;

grant execute on procedure delete\_movie\_from\_catalog to manager;

grant execute on procedure update\_hall\_def\_price to manager;

grant execute on procedure exportcountticketsfromtheatertoxml to manager;

grant execute on procedure generate\_timetable to manager;

grant execute on procedure importfilmsfromxml to manager;

grant execute on procedure update\_timetable\_entry to manager;

Листинг 2.2 – Предоставление привилегий

2.4 Вывод по разделу.

В данном разделе были представлены схема вариантов взаимодействия с базой данных, создание ролей и пользователей, а также предоставление пользователям привилегий.

# **3. Разработка необходимых объектов.**

3.1 Описание информационных объектов и ограничений целостности.

Схема базы данных ограничения целостности, связи и поля представлены на рисунке 3.1.

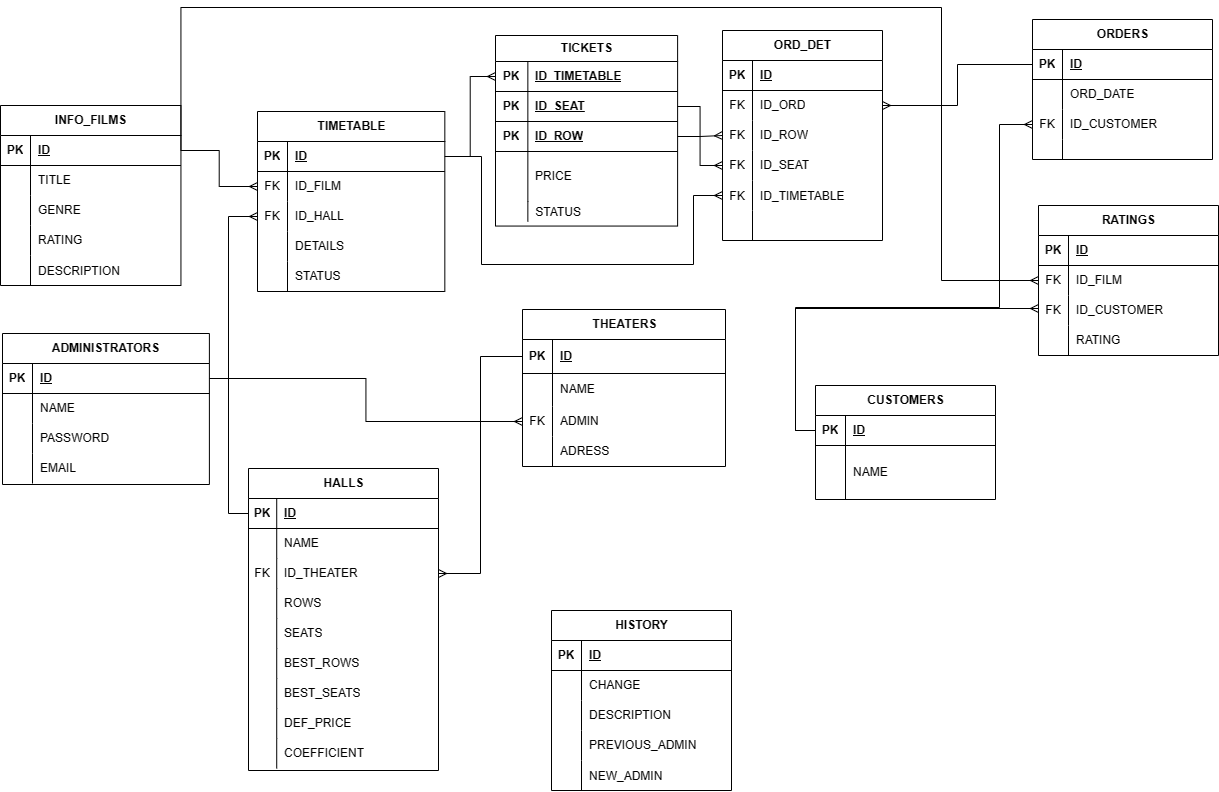


Рисунок 3.1 – Схема БД проекта

База данных была разработана на основе 11 таблиц.

3.1.1 Таблица INFO\_FILMS

В состав таблицы INFO\_FILMS входят следующие столбцы:

* + id\_film. Уникальный идентификатор фильма (SERIAL PRIMARY KEY)
  + title: Название фильма (VARCHAR(255), NOT NULL)
  + genre: Жанр фильма (VARCHAR(255), NOT NULL)
  + rating: Рейтинг фильма (FLOAT, NOT NULL)
  + description: Описание фильма (VARCHAR(2000))

3.1.2 Таблица RATINGS

В состав таблицы RATINGS входят следующие столбцы:

* + id\_rating: Уникальный идентификатор рейтинга (SERIAL PRIMARY KEY)
  + id\_film: Идентификатор фильма (INTEGER, FOREIGN KEY REFERENCES info\_films(id\_film))
  + id\_customer: Идентификатор покупателя (INTEGER, FOREIGN KEY REFERENCES customers(id\_customer))
  + rating: Оценка фильма (FLOAT, NOT NULL)

3.1.3 Таблица ADMINISTRATORS

В состав таблицы ADMINISTRATORS входят следующие столбцы:

* + id\_admin: Уникальный идентификатор администратора (SERIAL PRIMARY KEY)
  + name: Имя администратора (VARCHAR(255), NOT NULL)
  + password: Пароль администратора (VARCHAR(255), NOT NULL)
  + email: Электронная почта администратора (VARCHAR(255))

3.1.4 Таблица CUSTOMERS

В состав таблицы CUSTOMERS входят следующие столбцы:

* + id\_customer: Уникальный идентификатор покупателя (SERIAL PRIMARY KEY)
  + name: Имя покупателя (VARCHAR(255), NOT NULL)

3.1.5 Таблица THEATERS

В состав таблицы THEATERS входят следующие столбцы:

* + id\_theater: Уникальный идентификатор кинотеатра (SERIAL PRIMARY KEY)
  + name: Название кинотеатра (VARCHAR(255), NOT NULL)
  + address: Адрес кинотеатра (VARCHAR(255), NOT NULL)
  + admin: Идентификатор администратора кинотеатра (INTEGER, FOREIGN KEY REFERENCES administrators(id\_admin))

3.1.6 Таблица HALLS

В состав таблицы HALLS входят следующие столбцы:

* + id\_hall: Уникальный идентификатор зала (SERIAL PRIMARY KEY)
  + id\_theater: Идентификатор кинотеатра (INTEGER, FOREIGN KEY REFERENCES theaters(id\_theater))
  + name: Название зала (VARCHAR(255), NOT NULL)
  + rows: Количество рядов в зале (INTEGER, NOT NULL)
  + seats: Количество мест в ряду (INTEGER, NOT NULL)
  + def\_price: Цена по умолчанию (NUMERIC(10, 2), NOT NULL)
  + bestrows: Лучшие ряды (JSONB)
  + bestseats: Лучшие места (JSONB)
  + coefficient: Коэффициент, на который будет домнажаться цена по умолчанию (NUMERIC(10, 2), NOT NULL)

3.1.7 Таблица TIMETABLE

В состав таблицы TIMETABLE входят следующие столбцы:

* + id\_timetable: Уникальный идентификатор расписания (SERIAL PRIMARY KEY)
  + id\_film: Идентификатор фильма (INTEGER, FOREIGN KEY REFERENCES info\_films(id\_film))
  + id\_hall: Идентификатор зала (INTEGER, FOREIGN KEY REFERENCES halls(id\_hall))
  + details: Детали расписания (JSONB)
  + status: Статус расписания (VARCHAR(20))

3.1.8 Таблица ORDERS

В состав таблицы ORDERS входят следующие столбцы:

* + id\_ord: Уникальный идентификатор заказа (SERIAL PRIMARY KEY)
  + ord\_date: Дата заказа (TIMESTAMP without time zone)
  + id\_customer: Идентификатор покупателя (INTEGER, FOREIGN KEY REFERENCES customers(id\_customer))

3.1.9 Таблица ORD\_DET

В состав таблицы ORD\_DET входят следующие столбцы:

* + id\_ord\_det: Уникальный идентификатор деталей заказа (SERIAL PRIMARY KEY)
  + id\_ord: Идентификатор заказа (INTEGER, FOREIGN KEY REFERENCES orders(id\_ord))
  + id\_row: Идентификатор ряда (INTEGER)
  + id\_seat: Идентификатор места (INTEGER)
  + id\_timetable: Идентификатор расписания (INTEGER, FOREIGN KEY REFERENCES timetable(id\_timetable))

3.1.10 Таблица TICKETS

В состав таблицы TICKETS входят следующие столбцы:

* id\_row: Идентификатор ряда (SERIAL)
* id\_seat: Идентификатор места (SERIAL)
* id\_timetable: Идентификатор расписания (SERIAL, FOREIGN KEY REFERENCES timetable(id\_timetable))
* price: Цена билета (NUMERIC(10, 2))
* status: Статус билета (INTEGER, CHECK (status IN (0, 1)))

3.1.11 Таблица HISTORY

В состав таблицы HISTORY входят следующие столбцы:

* id\_change: Уникальный идентификатор изменения.
* change\_date: Дата и время изменения.
* description: Описание изменения.
* previous\_admin: Имя предыдущего администратора, совершившего изменение.
* new\_admin: Имя нового администратора, получившего права после изменения.

3.2 Процедуры.

Процедуры в базе данных являются именованными блоками кода, написанными на языке PL/pgSQL и предназначены для группировки и переиспользования кода, который может быть вызван несколько раз. Они представляют собой структуру, которая может содержать в себе различные типы операторов, такие как SQL-запросы, циклы, условные операторы, переменные и другие элементы.

Процедуры в базе данных могут быть использованы для выполнения различных задач, таких как обновление, удаление или выборка данных, а также для обработки данных внутри базы данных. Они могут быть вызваны из других процедур, функций, триггеров или приложений, что позволяет уменьшить нагрузку на сеть при обращении к базе данных.

Кроме того, процедуры могут содержать в себе параметры, которые могут быть переданы в качестве аргументов при вызове процедуры. Это позволяет создавать более гибкие и универсальные процедуры, которые могут быть использованы для выполнения различных задач с разными наборами данных.

Были разработаны следующие процедуры:

* generate\_timetable. Процедура для добавления сеанса
* delete\_movie\_from\_timetable. Процедура удаления сеанса
* update\_timetable\_entry. Процедура изменения существующего сеанса
* add\_movie\_to\_catalog. Процедура добавления фильма в каталог
* delete\_movie\_from\_catalog. Процедура удаления фильма из каталога
* add\_administrator. Процедура добавления администратора
* delete\_admin. Процедура удаления администратора
* update\_hall\_def\_price. Процедура изменения цены билета на сеанс
* add\_rating\_by\_title. Процедура добавления рейтинга на фильм
* buy\_ticket. Процедура покупки билета на сеанс
* return\_ticket. Процедура возврата билета на сеанс
* cancel\_ticket. Процедура для изменения количества доступных билетов на сеанс
* ExportCountTicketsFromTheaterToXML. Процедура для экспорта данных в XML-файл
* ImportFilmsFromXML. Процедура для импорта данных из XML-файла

|  |
| --- |
| CREATE OR REPLACE PROCEDURE add\_administrator(  admin\_name VARCHAR(255),  admin\_password VARCHAR(255),  admin\_email VARCHAR(255)  )SECURITY DEFINER  AS $$  BEGIN  IF EXISTS (SELECT 1 FROM administrators WHERE name = admin\_name) THEN  RAISE EXCEPTION 'Администратор с именем "%s" уже существует.', admin\_name;  END IF;    IF EXISTS (SELECT 1 FROM administrators WHERE email = admin\_email) THEN  RAISE EXCEPTION 'Администратор с почтой "%s" уже существует.', admin\_email;  END IF;  INSERT INTO administrators (name, password, email) VALUES (admin\_name, admin\_password, admin\_email);  RAISE NOTICE 'Администратор "%s" успешно добавлен.', admin\_name;  END;  $$ LANGUAGE plpgsql; |

Листинг 3.1 – Пример создания процедуры для добавления администратора

3.3 Функции

Функции являются объектами базы данных, которые позволяют выполнять определенные действия на стороне сервера базы данных. Функции могут использоваться для выполнения различных задач, таких как обработка данных, преобразование данных, агрегация данных и т. д.

Функции, разработанные в рамках курсового проекта:

* generate\_tickets. Генерация билетов на сеанс;
* delete\_related\_tickets. Удаление связанных с сеансом билетов;
* update\_tickets\_on\_timetable\_update. Изменение билетов на сеанс при изменении сеанса;
* search\_tickets\_by\_movie\_title. Поиск билетов по названию фильма;
* search\_tickets\_by\_session\_date. Поиск билетов по дате показа фильма;
* search\_tickets\_by\_genre. Поиск билетов по жанру фильма;
* sort\_films\_by\_rating. Сортировка фильмов по рейтингу;
* find\_tickets\_by\_quantity. Поиск билетов по количеству доступных;
* get\_available\_tickets\_summary. Просмотр количества доступных билетов на фильм;
* get\_tickets\_info. Просмотр стоимости билетов на фильмы;
* available\_movie\_tickets\_price. Просмотр количества билетов по стоимости;
* update\_average\_rating. Обновление рейтинга фильма;
* get\_tickets\_sold\_per\_theater. Просмотр количества проданных билетов в каждом кинотеатре;
* get\_user\_ticket\_purchases. Расчет итоговой стоимости набора билетов;
* add\_customer. Добавление покупателя;
* get\_theater\_movie\_tickets. Создание временной таблицы на основе представления и введенных дат для экспорта в XML-файл.

Пример функции songs\_for\_player, которая выводит все песни, представлена в листинге 3.3.

|  |
| --- |
| CREATE OR REPLACE FUNCTION find\_tickets\_by\_quantity(num\_tickets\_needed INTEGER)  RETURNS TABLE (  movie\_title VARCHAR(255),  film\_genre VARCHAR(255),  film\_rating FLOAT,  day DATE,  show\_time TIME,  available\_tickets BIGINT )  SECURITY DEFINER AS $$  BEGIN  RETURN QUERY  SELECT  f.title AS movie\_title,  f.genre AS film\_genre,  f.rating AS film\_rating,  (t.details->>'date')::DATE AS day,  (t.details->>'time')::TIME AS show\_time,  COUNT(\*) AS available\_tickets  FROM timetable t  INNER JOIN info\_films f ON t.id\_film = f.id\_film  LEFT JOIN tickets ti ON t.id\_timetable = ti.id\_timetable  AND ti.status = 1  GROUP BY f.title, f.genre, f.rating, (t.details->>'date')::DATE, (t.details->>'time')::TIME  HAVING COUNT(\*) >= num\_tickets\_needed;  END;  $$ LANGUAGE plpgsql; |

Листинг 3.2 – Пример создания функции для поиска билетов по количеству

3.4 Триггеры

Триггеры в PostgreSQL - это функции, которые автоматически вызываются при определенных событиях, происходящих в базе данных. Триггеры могут выполнять различные действия, например, проверять данные перед вставкой или изменением, обновлять связанные данные или записывать изменения в другие таблицы.

Триггеры, разработанные в рамках курсового проекта:

* generate\_tickets\_trigger. Триггер на добавление строк в таблицу TIMETABLE;
* delete\_related\_tickets\_trigger. Триггер на удаление строк из таблицы TIMETABLE;
* update\_tickets\_trigger. Триггер на обновление данных в таблцие TIMETABLE;
* update\_rating\_trigger. Триггер на добавление строк в таблице RATINGS;

Пример создания триггера update\_rating\_trigger представлен в листинге 3.4.

|  |
| --- |
| CREATE OR REPLACE FUNCTION update\_average\_rating()  RETURNS TRIGGER AS $$  BEGIN  UPDATE info\_films AS f  SET rating = (  SELECT AVG(rating)  FROM ratings  WHERE id\_film = NEW.id\_film  )  WHERE f.id\_film = NEW.id\_film;  RETURN NEW;  END;  $$ LANGUAGE plpgsql;  CREATE TRIGGER update\_rating\_trigger  AFTER INSERT ON ratings  FOR EACH ROW  EXECUTE FUNCTION update\_average\_rating(); |

Листинг 3.3 – Пример создания функции для добавления рейтинга

3.5 Представление

Представление – виртуальная таблица, которая основана на результатах выполнения запроса. Оно является логической оболочкой над одним или несколькими базовыми таблицами, обеспечивая удобный способ абстрагирования и доступа к данным.

Было разработано следующее представление:

* user\_ticket\_purchases. Представление, содержащее покупателей, дату покупки и итоговую стоимость набора билетов.

|  |
| --- |
| CREATE VIEW user\_ticket\_purchases AS  SELECT  c.name AS user\_email,  o.ord\_date AS purchase\_date,  SUM(t.price) AS total\_ticket\_price  FROM  orders o  JOIN  ord\_det od ON o.id\_ord = od.id\_ord  JOIN  tickets t ON od.id\_row = t.id\_row AND od.id\_seat = t.id\_seat AND od.id\_timetable = t.id\_timetable  JOIN  customers c ON o.id\_customer = c.id\_customer  GROUP BY  c.name, o.ord\_date; |

Листинг 3.4 – Пример создания представления набора билетов покупателей

3.6 Вывод

В данном разделе была описана проектируемая база данных, все таблицы базы данных и ограничения целостности к строкам таблицы. Также была определена информация, которая будет хранится в таблицах. В рамках разработки модели базы данных были созданы следующие объекты:

11 таблиц, содержащих данные о фильмах, кинотеатрах, залах и др.

14 процедур для выполнения различных операций с данными, таких как добавление фильмов в каталог, добавление администраторов, покупка билетов и др.

14 функций для получения различной информации из базы данных, например, поиск билетов по различным параметрам, сортировка фильмов по рейтингу, добавление пользователей.

4 триггера для автоматического добавления, изменения и удаления данных в таблицах.

1. Описание процедур экспорта и импорта.

**4.1 Процедура экспорта данных**

XML — это формат, который хранит структурированную информацию и в основном используется для передачи данных между сервером и клиентом.

Часто возникает необходимость импортировать и экспортировать XML-файлы, в данной курсовой работе используются процедура ExportCountTicketsFromTheaterToXML() для экспорта. Процедура экспорта используются для временной таблицы temp\_theater\_movie\_tickets, так как эта таблица является необходимой в базе данных. Создания процедуры можно посмотреть в листинге 4.1.

|  |
| --- |
| CREATE OR REPLACE PROCEDURE ExportCountTicketsFromTheaterToXML(  file\_path TEXT  ) AS $$  DECLARE  xml\_data TEXT := '<?xml version="1.0" encoding="UTF-8"?><TheatersTickets>';  theater\_movie\_ticket\_rec RECORD;  BEGIN  FOR theater\_movie\_ticket\_rec IN SELECT \* FROM temp\_theater\_movie\_tickets LOOP  xml\_data := xml\_data || '<theater\_movie\_ticket>';  xml\_data := xml\_data || '<theater\_name>' || theater\_movie\_ticket\_rec.theater\_name || '</theater\_name>';  xml\_data := xml\_data || '<movie\_title>' || theater\_movie\_ticket\_rec.movie\_title || '</movie\_title>';  xml\_data := xml\_data || '<total\_tickets\_sold>' || theater\_movie\_ticket\_rec.total\_tickets\_sold || '</total\_tickets\_sold>';  xml\_data := xml\_data || '<from\_date>' || theater\_movie\_ticket\_rec.first\_purchase\_date || '</from\_date>';  xml\_data := xml\_data || '<to\_date>' || theater\_movie\_ticket\_rec.last\_purchase\_date || '</to\_date>';  xml\_data := xml\_data || '</theater\_movie\_ticket>';  END LOOP;  xml\_data := xml\_data || '</TheaterTickets>';  EXECUTE format('COPY (SELECT %L) TO %L', xml\_data, file\_path);  RAISE NOTICE 'Данные успешно загружены в XML файл: %', file\_path;  EXCEPTION  WHEN OTHERS THEN  RAISE NOTICE 'Ошибка при экспортировке данных в XML: %', SQLERRM;  END;  $$ LANGUAGE plpgsql; |

Листинг 4.1 – Пример создания процедуры экспорта в xml файл

**4.2 Процедура импорта данных**

Также возникает необходимость импортировать XML-файлы в базу данных, в данной курсовой работе используются процедура ImportFilmsFromXML() для импорта. Пример создания процедуры можно посмотреть в листинге 4.2.

|  |
| --- |
| CREATE OR REPLACE PROCEDURE ImportFilmsFromXML(file\_path VARCHAR)  security definer  AS $$  DECLARE  xml\_data TEXT;  BEGIN  -- Read data from the file  xml\_data := pg\_read\_file(file\_path);  -- Check if data is read successfully  IF xml\_data IS NULL THEN  RAISE EXCEPTION 'Failed to read data from file %', file\_path;  END IF;  -- Display the data read from the file for debugging  RAISE INFO 'Данные проверяются из файла: %', xml\_data;  -- Create a temporary table for importing data  CREATE TEMP TABLE tmp\_films (  title VARCHAR(255),  genre VARCHAR(255),  rating INT,  description VARCHAR(2000)  );  -- Insert new data from XML into the temporary table  BEGIN  EXECUTE 'INSERT INTO tmp\_films (title, genre, rating, description)  SELECT  unnest(xpath(''/info\_films/info\_films/title/text()'',  xmlparse(document ''' || xml\_data || ''')))::text AS title,  unnest(xpath(''/info\_films/info\_films/genre/text()'',  xmlparse(document ''' || xml\_data || ''')))::text AS genre,  CAST(TRIM(unnest(xpath(''/info\_films/info\_films/rating/text()'',  xmlparse(document ''' || xml\_data || ''')))::text) AS INT) AS rating,  unnest(xpath(''/info\_films/info\_films/description/text()'',  xmlparse(document ''' || xml\_data || ''')))::text AS description';  EXCEPTION  WHEN OTHERS THEN  RAISE EXCEPTION 'Возникла ошибка с импортом данных из XML: %', SQLERRM;  END;  -- Insert data from the temporary table into info\_films  INSERT INTO info\_films (title, genre, rating, description)  SELECT title, genre, rating, description FROM tmp\_films;  RAISE INFO 'Данный каталог был успешно перенесен из файла % во временную таблицу tmp\_films', file\_path;  END;  $$ LANGUAGE plpgsql; |

Листинг 4.2 – Пример создания процедуры импорта из xml файла

4.3 Вывод

В данном разделе были описаны примеры разработанных процедур импорта и экспорта данных. Также были продемонстрированы листинги кода процедур.

# **5. Тестирование производительности базы данных.**

5.1 Тестирование производительности по таблице TICKETS.

Производительность БД является решающим фактором эффективности управленческих и коммерческих приложений. Если поиск или запись данных выполняется медленно – способность к нормальной работе приложения падает. Существует единственный путь выяснить причину плохой производительности – выполнить количественные измерения и определить, что является причиной проблемы производительности.

Для того чтобы правильно организовать процесс тестирования БД, тестировщики должны обладать хорошими знаниями SQL и DML и иметь ясное представление о внутренней структуре БД. Это самый лучший и надежный способ тестирования БД особенно для приложений с низким и средним уровнем сложности. Данный метод не только дает уверенность, что тестирование выполнено качественно, но также повышает мастерство написания SQL-запросов.

В PostgreSQL оптимизация запросом в основном заключается в построение индексов над таблицами.

Для тестирования производительности были добавлены 100 000 записей в таблицу TICKETS.

Ниже на рисунке 4.1 представлен результат Select-запроса с секцией WHERE к таблице до добавления индекса. Время выполнения составляет 165 мс.

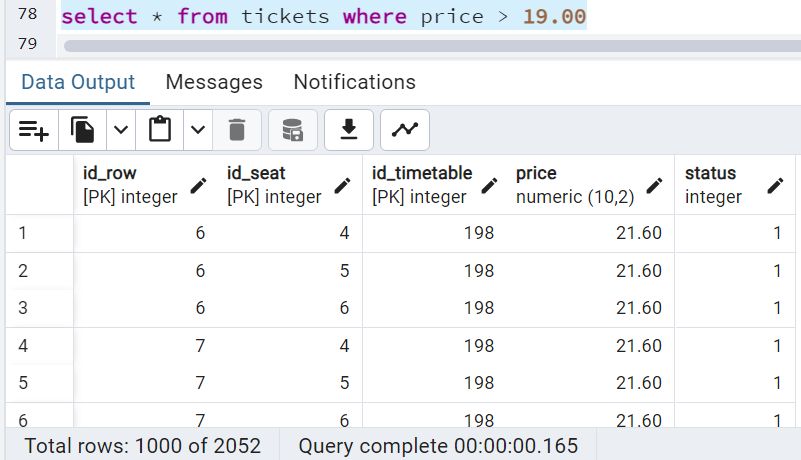


Рисунок 4.1 – Результат Select-запроса к таблице без индекса

На рисунке 4.2 представлен результат Select-запроса к таблице после добавления индекса. Время выполнения уже составляет 87 мс.

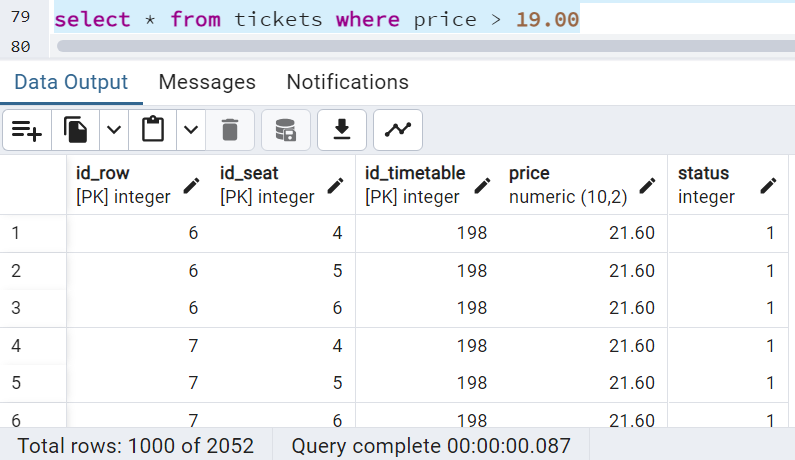


Рисунок 4.2 – Результат Select-запроса к таблице с индексом

На рисунке 4.3 представлен результат Select-запроса с секцией ORDER BY к таблице до добавления индекса. Время выполнения составляет 212 мс.

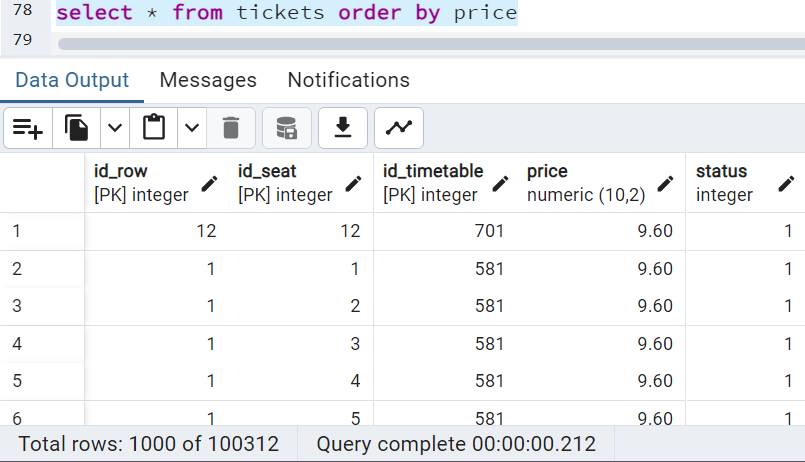


Рисунок 4.3 – Результат Select-запроса к таблице без индекса

На рисунке 4.4 представлен результат Select-запроса к таблице после добавления индекса. Время выполнения уже составляет 127 мс.

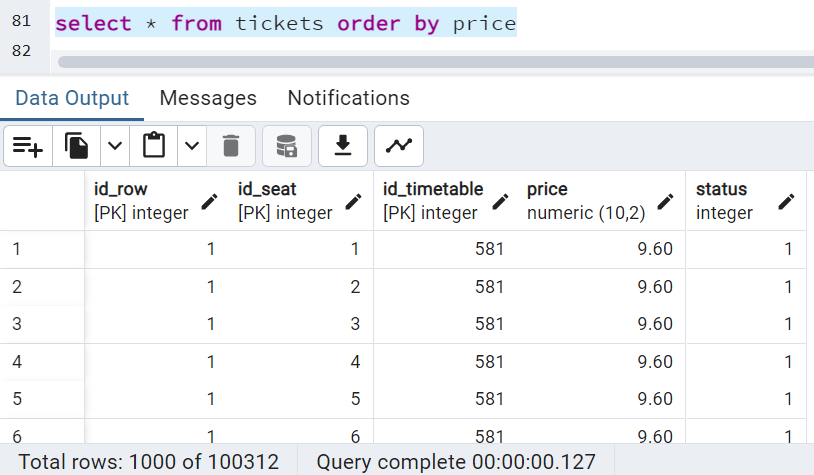


Рисунок 4.4 – Результат Select-запроса к таблице с индексом

На рисунке 4.5 представлен результат Select-запроса с использованием JOIN до добавления индекса. Время выполнения составляет 615 мс.

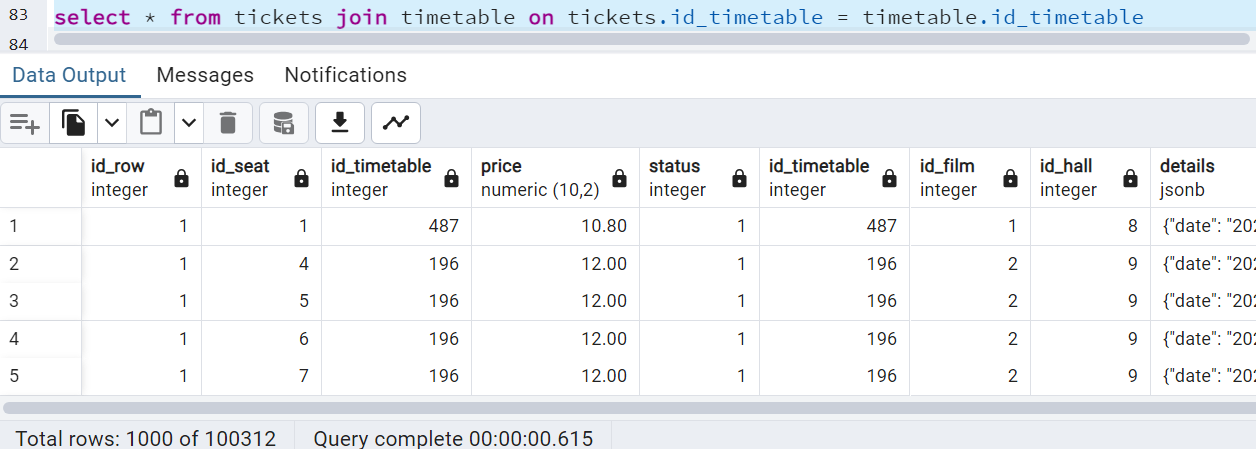


Рисунок 4.5 – Результат Select-запроса к таблице без индекса

На рисунке 4.6 представлен результат Select-запроса к таблице после добавления индекса. Время выполнения уже составляет 359 мс.

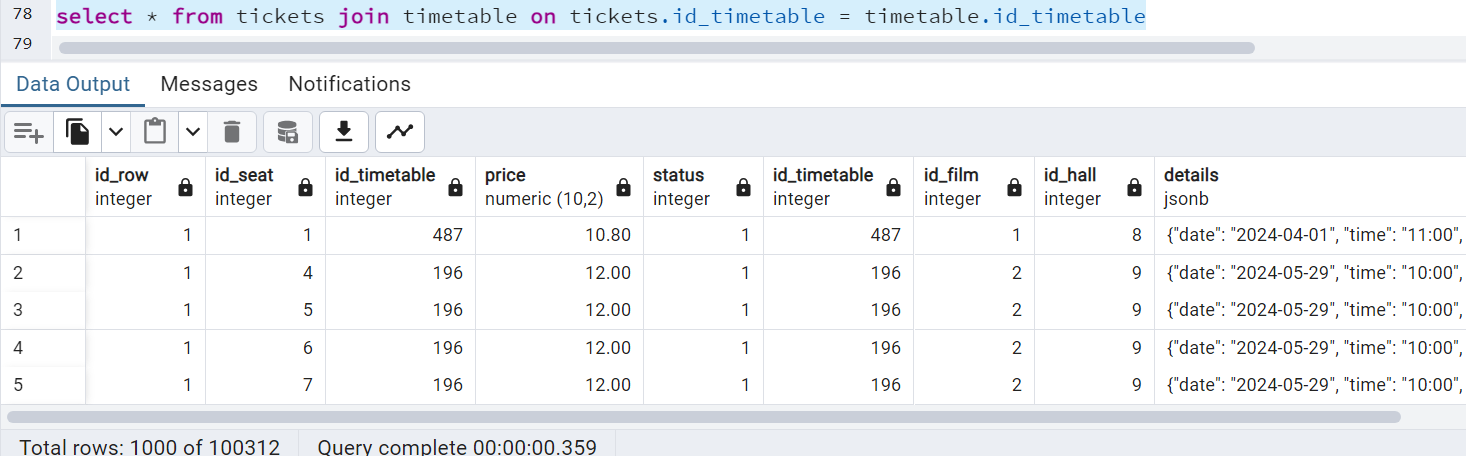


Рисунок 4.6 – Результат Select-запроса к таблице с индексом

5.2 Создание индекса

Индексы являются структурами данных, которые ускоряют выполнение запросов на поиск, сортировку или группировку данных в таблицах. Индексы создаются на основе значений столбцов в таблице и предоставляют быстрый доступ к записям, удовлетворяющим определенным условиям.

Индекс, разработанный в рамках курсового проекта, для оптимизации запросов:

* idx\_tickets\_price.

Создания индекса представлено в листинге 5.1

create index idx\_tickets\_price on tickets (price)

Листинг 5.1 – Создание индекса idx\_tickets\_price

5.3 Вывод.

В данном разделе было описано тестирование производительности разрабатываемой базы данных. Были выполнены select-запросы с различными операторами и продемонстрировано время их выполнения до и после добавления индекса. Также было описано создание индекса для оптимизации.

# **6.** **Описание технологии и ее применение в базе данных**

6.1 Технология Full Text Search

Полнотекстовый поиск (Full-Text Search) — это технология, используемая для выполнения поиска и анализа текстовых данных, учитывающая семантическое значение слов и их контекст. Она позволяет эффективно находить соответствия между запросами пользователей и текстовыми документами, основываясь на содержимом текста, а не только на точном совпадении слов. Тип tsvector представляет собой что-то вроде нормализованной строки, по которой будет производиться поиск. Под нормализацией понимается выкидывание стоп-слов, таких, как предлоги, обрезание окончаний слов, и так далее.

Для создания технологии необходимо было установить два расширения pg\_trgm и unaccent.

Pg\_trgm (trigram) - это расширение, которое добавляет поддержку треграммного поиска в PostgreSQL. Треграммы - это последовательности из трех символов, используемые для сравнения и поиска текстовых данных. Расширение pg\_trgm позволяет создавать индексы на треграммах и выполнять поиск по сходству строк на основе треграмм. Это полезно, когда необходимо находить сходство между строками, даже если они не совпадают точно.

Unaccent - это расширение, которое обеспечивает возможность удаления диакритических знаков (акцентов) из текстовых данных. Некоторые языки используют акцентированные символы, и иногда может возникнуть необходимость в поиске и сравнении текста без учета акцентов. Расширение unaccent предоставляет функцию unaccent(), которая позволяет удалить акценты и сравнивать текст без них.

Добавление этих расширений в проект показано на рисунке 6.1.

create extension if not exists pg\_trgm

create extension if not exists unaccent

Листинг 6.1 – Подключение расширений pg\_trgm и unaccent

После подключения этих расширений мы можем использовать доступные в PostgreSQL операторы и функции.

Одна из интересных особенностей полнотекстового поиска – фразовый поиск. Фразовый поиск используется для поиска фразы, содержащей несколько слов в определенном порядке. Для ускорения поиска применяются GIN-индексы. Они обычно используются для близких совпадений или совпадений по расстоянию. Реализация индекса представлена на листинге 6.2.

CREATE INDEX idx\_description\_fts

ON info\_films

USING gin(to\_tsvector('russian', description));

Листинг 6.2 – Создание GIN-индекса

PostgreSQL поддерживает расширенный синтаксис запросов полнотекстового поиска. Можно использовать функции, такие to\_tsvector, websearch\_to\_tsquery, ts\_rank или операторы, например, оператор @@. Реализация функций и оператора представлена на листинге 6.3.

SELECT \*,

ts\_rank(to\_tsvector('russian', description), websearch\_to\_tsquery('russian', 'супер фильма')) AS rank

FROM info\_films

WHERE to\_tsvector('russian', description) @@ websearch\_to\_tsquery('russian', 'супер фильма')

ORDER BY rank DESC;

Листинг 6.3 – Реализация функций и оператора

Еще одним примером функции является plainto\_tsquery - эта функция, которая преобразует простой текстовый запрос в формат, понятный для выполнения полнотекстового поиска. Она принимает текстовую строку и возвращает специальный тип данных tsquery, который может использоваться для сравнения с полнотекстовыми векторами. Пример использования представлен на листинге 6.4.

SELECT \*,

ts\_rank(to\_tsvector('russian', description), plainto\_tsquery('russian', 'Миллиардеры')) AS rank

FROM info\_films

WHERE to\_tsvector('russian', description) @@ plainto\_tsquery('russian', 'Миллиардеры')

ORDER BY rank DESC;

Листинг 6.4 – Реализация функции plainto\_tsquery

Синтаксис полнотекстового поиска отличается в различных СУБД, например, известный оператор MATCH-AGAINST не является частью PostgreSQL, но его аналогом является оператор @@. Оператор @@ возвращает true, если вектор соответствует запросу. Применение данного оператора представлена в листингах 6.3 и 6.4.

6.2 Вывод

Полнотекстовый поиск позволяет искать информацию, используя естественный язык, и получать наиболее релевантные результаты, даже если запрос не точно соответствует искомым словам.

# **7. Сценарий использования.**

**7.1 Сценария использования для управляющего**.

Сценарий работы с базой данных (БД) представляет собой последовательность шагов или действий, необходимых для взаимодействия с БД. Этот сценарий описывает типичные операции, которые могут выполняться с БД.

Для данного проекта был разработан сценарий, который отражает функциональные возможности каждого пользователя. Рисунок сценария представлен на рисунке 7.1.

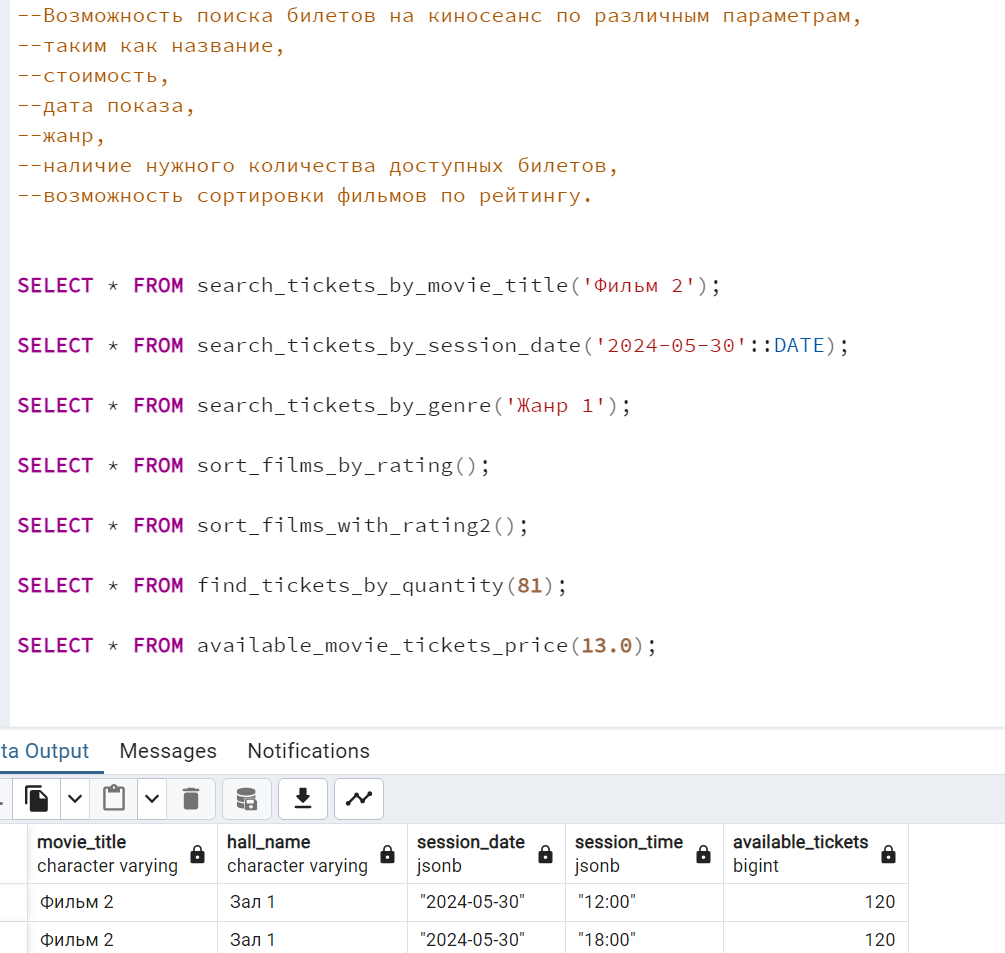


Рисунок 7.1 – Сценарий к базе данных

Ниже продемонстрирована работоспособность функции и процедуры.

Функция search\_tickets\_by\_session\_date представлена на рисунке7.2.

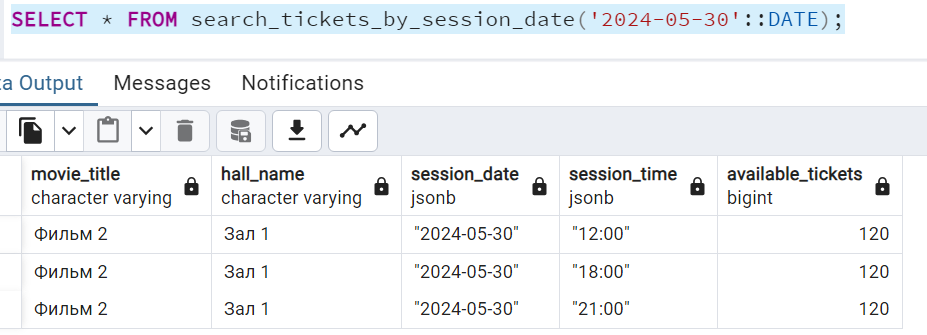


Рисунок 7.2 – Демонстрация функции search\_tickets\_by\_session\_date

Процедура add\_movie\_to\_catalog представлена на рисунке 7.3.

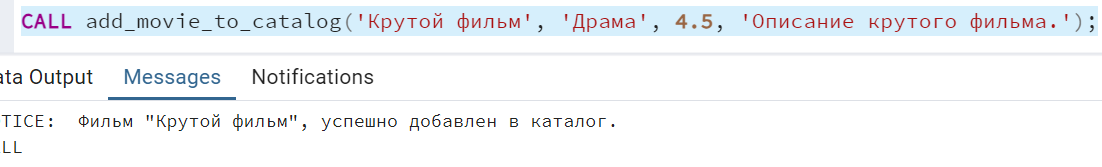


Рисунок 7.3 – Демонстрация процедуры add\_movie\_to\_catalog

7.2 Вывод

В данном разделе были представлены функция и процедура, подтверждающая работоспособность базы данных, а также продеменстрирован сценарий.

# **Заключение**

В процессе выполнения курсовой работы была достигнута поставленная цель по созданию базы данных. Удалось обеспечить гибкость и надежность базы данных за счет оптимизации структуры таблиц и исключения избыточности. Была реализована инкапсуляция внутренней структуры базы данных посредством создания пользователей и ролей. Также был выполнен анализ производительности базы данных и принято решение создать индекс, что существенно улучшило скорость доступа к данным. При разработке проекта выполнены следующие пункты:

* поиск билетов на киносеансы с использованием различных параметров, такие как название фильма, стоимость, дата показа, жанр, наличие доступных билетов;
* возможность сортировки фильмов по рейтингу;
* добавление, изменение и удаление сеансов;
* возможность изменения списка фильмов в прокате, регулирования цены билета на сеанс и добавления/удаления администраторов;
* возможность добавления данных о покупателях, расчета общей стоимости набора билетов и количества проданных билетов, изменения количества доступных;
* возможность просмотра количества доступных билетов, их стоимость, покупки и возврата билетов, а также оценки фильма;
* экспорт и импорт данных в формат XML;
* реализация технологии полнотекстового поиска;
* проверка работоспособности и производительности системы на данных большого объема.

В результате выполнения курсового проекта были получены навыки по взаимодействию с базой данных PostgreSQL, изучена технология полнотекстового поиска, а также использование различных типов данных и форматов обмена информацией. В соответствии с полученным результатом работы системы можно сделать вывод, что разработанная программа работает корректно, и требования технического задания выполнены в полном объеме.

**Список использованных литературных источников**

1. Skyline [Электронный ресурс] / Режим доступа: https://skyline.com/– Дата доступа: 29.03.2024.
2. Moоon [Электронный ресурс] / Режим доступа: https://mooon.by/ – Дата доступа: 29.03.2024.
3. ProfessorWeb .NET & Web Programming [Электронный ресурс] / Режим доступа: https://professorweb.ru Дата доступа: 24.04.2024.
4. Microsoft Docs Archived Content [Электронный ресурс]/ Режим доступа: https://docs.microsoft.com/en-us/archive/ Дата доступа: 20.04.2024.
5. Форум для программистов или разработчиков [Электронный ресурс] – https://stackoverflow.com/ – Дата доступа: 25.04.2024.
6. Постгрес: проектирование и реализация БД / ред. С. Л. Шумский. – СПб.: БХВ-Петербург, 2009. - 287 с.

# **Приложение А**

Листинг таблиц

CREATE TABLE info\_films (

id\_film SERIAL PRIMARY KEY,

title VARCHAR(255) NOT NULL,

genre VARCHAR(255) NOT NULL,

rating FLOAT NOT NULL,

description VARCHAR(2000)

);

Листинг А.1 – Таблица INFO\_FILMS

CREATE TABLE ratings (

id\_rating SERIAL PRIMARY KEY,

id\_film INTEGER REFERENCES info\_films (id\_film),

id\_customer INTEGER REFERENCES customers (id\_customer),

rating FLOAT NOT NULL

);

Листинг А.2 – Таблица RATINGS

CREATE TABLE administrators (

id\_admin SERIAL PRIMARY KEY,

name VARCHAR(255) NOT NULL,

password VARCHAR(255) NOT NULL,

email VARCHAR(255)

);

Листинг А.3 – Таблица ADMINISTRATORS

CREATE TABLE customers (

id\_customer SERIAL PRIMARY KEY,

name VARCHAR(255) NOT NULL

);

Листинг А.4 – Таблица CUSTOMERS

CREATE TABLE theaters (

id\_theater SERIAL PRIMARY KEY,

name VARCHAR(255) NOT NULL,

address VARCHAR(255) NOT NULL,

admin serial REFERENCES administrators (id\_admin)

);

Листинг А.5 – Таблица THEATERS

CREATE TABLE halls (

id\_hall SERIAL PRIMARY KEY,

id\_theater INTEGER REFERENCES theaters (id\_theater),

name VARCHAR(255) NOT NULL,

rows integer not null,

seats integer not null,

def\_price NUMERIC(10, 2) NOT NULL,

bestrows jsonb,

bestseats jsonb,

coefficient NUMERIC(10, 2) not null

);

Листинг А.6 – Таблица HALLS

CREATE TABLE timetable (

id\_timetable SERIAL PRIMARY KEY,

id\_film INTEGER REFERENCES info\_films (id\_film),

id\_hall INTEGER REFERENCES halls (id\_hall),

details JSONB,

status VARCHAR(20)

);

Листинг А.7 – Таблица TIMETABLE

CREATE TABLE orders (

id\_ord SERIAL PRIMARY KEY,

ord\_date TIMESTAMP without time zone,

id\_customer INTEGER REFERENCES customers (id\_customer)

);

Листинг А.8 – Таблица ORDERS

CREATE TABLE ord\_det (

id\_ord\_det SERIAL PRIMARY KEY,

id\_ord INTEGER REFERENCES orders (id\_ord),

id\_row INTEGER,

id\_seat INTEGER,

id\_timetable INTEGER REFERENCES timetable (id\_timetable),

FOREIGN KEY (id\_row, id\_seat, id\_timetable) REFERENCES tickets (id\_row, id\_seat, id\_timetable)

);

Листинг А.9 – Таблица ORD\_DET

CREATE TABLE tickets (

id\_row SERIAL,

id\_seat SERIAL,

id\_timetable SERIAL REFERENCES timetable (id\_timetable),

price NUMERIC(10, 2),

status INTEGER CHECK (status IN (0, 1)),

PRIMARY KEY (id\_row, id\_seat, id\_timetable)

);

Листинг А.10 – Таблица TICKETS

CREATE TABLE history (

id\_change SERIAL PRIMARY KEY,

change\_date DATE,

description VARCHAR(255),

previous\_admin VARCHAR(255),

new\_admin VARCHAR(255)

);

Листинг А.11 – Таблица HISTORY

# **Приложение Б**

Листинг процедур

CREATE OR REPLACE PROCEDURE generate\_timetable(

film\_title VARCHAR(100),

hall\_name VARCHAR(100),

start\_date DATE,

end\_date DATE,

time\_coefficients JSONB,

status VARCHAR(20)

)

AS $$

DECLARE

film\_id INTEGER;

hall\_id INTEGER;

curr\_date DATE;

start\_time TIME;

time\_coefficient FLOAT;

details\_json JSONB;

BEGIN

-- Проверка существования фильма в таблице info\_films

SELECT id\_film INTO film\_id FROM info\_films WHERE title = film\_title;

IF NOT FOUND THEN

RAISE EXCEPTION 'Фильм с названием "%", указанным в параметрах, не найден в базе данных.', film\_title;

END IF;

-- Проверка существования зала в таблице halls

SELECT id\_hall INTO hall\_id FROM halls WHERE name = hall\_name;

IF NOT FOUND THEN

RAISE EXCEPTION 'Зал с названием "%", указанным в параметрах, не найден в базе данных.', hall\_name;

END IF;

curr\_date := start\_date;

-- Цикл по датам

WHILE curr\_date <= end\_date LOOP

-- Парсинг временных коэффициентов

FOR i IN 0..jsonb\_array\_length(time\_coefficients)-1 LOOP

start\_time := (time\_coefficients->i->>'time')::TIME;

time\_coefficient := (time\_coefficients->i->>'coefficient')::FLOAT;

-- Создание JSONB-объекта для поля details

details\_json := jsonb\_build\_object('date', to\_char(curr\_date, 'YYYY-MM-DD'), 'time', to\_char(start\_time, 'HH24:MI'), 'coefficient', time\_coefficient);

-- Вставка записи в таблицу "Расписание"

INSERT INTO timetable (id\_film, id\_hall, details, status)

VALUES (film\_id, hall\_id, details\_json, status);

END LOOP;

curr\_date := curr\_date + 1; -- Переход к следующей дате

END LOOP;

RAISE NOTICE 'Расписание успешно сгенерировано.';

EXCEPTION

WHEN OTHERS THEN

RAISE NOTICE 'Ошибка: %', SQLERRM;

END;

$$ LANGUAGE plpgsql;

Листинг Б.1 – процедура generate\_timetable

CREATE OR REPLACE PROCEDURE delete\_movie\_from\_timetable(

film\_title VARCHAR(100),

hall\_name VARCHAR(100),

theater\_name VARCHAR(100)

)

AS $$

DECLARE

film\_id INTEGER;

hall\_id INTEGER;

theater\_id INTEGER;

BEGIN

-- Получение идентификатора фильма по названию

SELECT id\_film INTO film\_id FROM info\_films WHERE title = film\_title;

-- Получение идентификатора зала и кинотеатра по их названиям

SELECT h.id\_hall, t.id\_theater INTO hall\_id, theater\_id

FROM halls h

INNER JOIN theaters t ON h.id\_theater = t.id\_theater

WHERE h.name = hall\_name AND t.name = theater\_name;

-- Проверка наличия записи о фильме в заданном зале и кинотеатре

IF NOT EXISTS (

SELECT 1 FROM timetable

WHERE id\_film = film\_id AND id\_hall = hall\_id

) THEN

RAISE NOTICE 'Некорректные данные. Фильм "%s" не показывается в зале "%s" кинотеатра "%s".', film\_title, hall\_name, theater\_name;

RETURN;

END IF;

-- Удаление связанных билетов из таблицы "tickets"

DELETE FROM tickets

WHERE id\_timetable IN (

SELECT id\_timetable FROM timetable

WHERE id\_film = film\_id AND id\_hall = hall\_id

);

-- Удаление записи из таблицы "timetable" по заданным условиям

DELETE FROM timetable

WHERE id\_film = film\_id AND id\_hall = hall\_id;

RAISE NOTICE 'Запись о фильме "%", показываемом в зале "%", кинотеатра "%", успешно удалена из расписания.', film\_title, hall\_name, theater\_name;

END;

$$ LANGUAGE plpgsql;

Листинг Б.2 – процедура delete\_movie\_from\_timetable

CREATE OR REPLACE PROCEDURE update\_timetable\_entry(

film\_title VARCHAR(100),

hall\_name VARCHAR(100),

start\_date DATE,

end\_date DATE,

time\_coefficients JSONB,

status VARCHAR(20)

)

AS $$

DECLARE

film\_id INTEGER;

hall\_id INTEGER;

curr\_date DATE;

start\_time TIME;

time\_coefficient FLOAT;

details\_json JSONB;

BEGIN

-- Проверка существования фильма в таблице info\_films

SELECT id\_film INTO film\_id FROM info\_films WHERE title = film\_title;

IF NOT FOUND THEN

RAISE EXCEPTION 'Фильм с названием "%", указанным в параметрах, не найден в базе данных.', film\_title;

END IF;

-- Проверка существования зала в таблице halls

SELECT id\_hall INTO hall\_id FROM halls WHERE name = hall\_name;

IF NOT FOUND THEN

RAISE EXCEPTION 'Зал с названием "%", указанным в параметрах, не найден в базе данных.', hall\_name;

END IF;

curr\_date := start\_date;

DELETE FROM tickets

WHERE id\_timetable IN (

SELECT id\_timetable FROM timetable

WHERE id\_film = film\_id AND id\_hall = hall\_id

AND details->>'date' >= to\_char(start\_date, 'YYYY-MM-DD')

AND details->>'date' <= to\_char(end\_date, 'YYYY-MM-DD')

);

-- Удаление существующих записей в таблице "Расписание" для указанного фильма, зала и периода

DELETE FROM timetable

WHERE id\_film = film\_id AND id\_hall = hall\_id AND details->>'date' >= to\_char(start\_date, 'YYYY-MM-DD') AND details->>'date' <= to\_char(end\_date, 'YYYY-MM-DD');

-- Цикл по датам

WHILE curr\_date <= end\_date LOOP

-- Парсинг временных коэффициентов

FOR i IN 0..jsonb\_array\_length(time\_coefficients)-1 LOOP

start\_time := (time\_coefficients->i->>'time')::TIME;

time\_coefficient := (time\_coefficients->i->>'coefficient')::FLOAT;

-- Создание JSONB-объекта для поля details

details\_json := jsonb\_build\_object('date', to\_char(curr\_date, 'YYYY-MM-DD'), 'time', to\_char(start\_time, 'HH24:MI'), 'coefficient', time\_coefficient);

-- Вставка записи в таблицу "Расписание"

INSERT INTO timetable (id\_film, id\_hall, details, status)

VALUES (film\_id, hall\_id, details\_json, status);

END LOOP;

curr\_date := curr\_date + 1; -- Переход к следующей дате

END LOOP;

RAISE NOTICE 'Расписание успешно обновлено.';

EXCEPTION

WHEN OTHERS THEN

RAISE NOTICE 'Ошибка: %', SQLERRM;

END;

$$ LANGUAGE plpgsql;

Листинг Б.3 – процедура update\_timetable\_entry

CREATE OR REPLACE PROCEDURE add\_movie\_to\_catalog(

movie\_title VARCHAR(255),

movie\_genre VARCHAR(255),

movie\_rating FLOAT,

movie\_description VARCHAR(2000)

)

security definer

AS $$

BEGIN

IF EXISTS (SELECT 1 FROM info\_films WHERE title = movie\_title) THEN

RAISE EXCEPTION 'Фильм с названием "%", уже присутствует в каталоге.', movie\_title;

END IF;

INSERT INTO info\_films (title, genre, rating, description)

VALUES (movie\_title, movie\_genre, movie\_rating, movie\_description);

RAISE NOTICE 'Фильм "%", успешно добавлен в каталог.', movie\_title;

END;

$$ LANGUAGE plpgsql;

Листинг Б.4 – процедура add\_movie\_to\_catalog

CREATE OR REPLACE PROCEDURE delete\_movie\_from\_catalog(

film\_title VARCHAR(255)

)

security definer

AS $$

BEGIN

IF NOT EXISTS (SELECT 1 FROM info\_films WHERE title = film\_title) THEN

RAISE EXCEPTION 'Фильм "%" не найден в каталоге.', film\_title;

END IF;

DELETE FROM info\_films

WHERE title = film\_title;

RAISE NOTICE 'Фильм "%", успешно удален из каталога.', film\_title;

END;

$$ LANGUAGE plpgsql;

Листинг Б.5 – процедура delete\_movie\_from\_catalog

CREATE OR REPLACE PROCEDURE add\_administrator(

admin\_name VARCHAR(255),

admin\_password VARCHAR(255),

admin\_email VARCHAR(255)

)

security definer

AS $$

BEGIN

IF EXISTS (SELECT 1 FROM administrators WHERE name = admin\_name) THEN

RAISE EXCEPTION 'Администратор с именем "%s" уже существует.', admin\_name;

END IF;

IF EXISTS (SELECT 1 FROM administrators WHERE email = admin\_email) THEN

RAISE EXCEPTION 'Администратор с почтой "%s" уже существует.', admin\_email;

END IF;

INSERT INTO administrators (name, password, email) VALUES (admin\_name, admin\_password, admin\_email);

RAISE NOTICE 'Администратор "%s" успешно добавлен.', admin\_name;

END;

$$ LANGUAGE plpgsql;

Листинг Б.6 – процедура add\_administrator

CREATE OR REPLACE PROCEDURE update\_hall\_def\_price(

p\_hall\_name VARCHAR(255),

p\_theater\_name VARCHAR(255),

p\_new\_def\_price NUMERIC(10, 2)

)

security definer

AS $$

DECLARE

hall\_id INTEGER;

theater\_id INTEGER;

BEGIN

-- Получаем id кинотеатра по его названию

SELECT id\_theater INTO theater\_id

FROM theaters

WHERE name = p\_theater\_name;

-- Проверяем, был ли найден кинотеатр

IF theater\_id IS NULL THEN

RAISE EXCEPTION 'Кинотеатр с названием "%" не найден.', p\_theater\_name;

END IF;

-- Получаем id зала по его названию и id кинотеатра

SELECT id\_hall INTO hall\_id

FROM halls

WHERE name = p\_hall\_name

AND id\_theater = theater\_id;

-- Проверяем, был ли найден зал

IF hall\_id IS NULL THEN

RAISE EXCEPTION 'Зал с названием "%" в кинотеатре "%" не найден.', p\_hall\_name, p\_theater\_name;

END IF;

-- Обновляем значение def\_price

UPDATE halls

SET def\_price = p\_new\_def\_price

WHERE id\_hall = hall\_id

AND id\_theater = theater\_id;

-- Выводим сообщение об успешном обновлении

RAISE NOTICE 'Значение def\_price для зала "%" в кинотеатре "%" успешно обновлено на %.', p\_hall\_name, p\_theater\_name, p\_new\_def\_price;

END;

$$ LANGUAGE plpgsql;

Листинг Б.7 – процедура update\_hall\_def\_price

CREATE OR REPLACE PROCEDURE delete\_admin (

admin\_name VARCHAR(255),

admin\_password VARCHAR(255),

admin\_email VARCHAR(255)

)

security definer

AS $$

BEGIN

IF EXISTS (SELECT 1 FROM administrators WHERE name = admin\_name) THEN

RAISE EXCEPTION 'Администратор с именем "%s" уже существует.', admin\_name;

END IF;

IF EXISTS (SELECT 1 FROM administrators WHERE email = admin\_email) THEN

RAISE EXCEPTION 'Администратор с почтой "%s" уже существует.', admin\_email;

END IF;

INSERT INTO administrators (name, password, email) VALUES (admin\_name, admin\_password, admin\_email);

RAISE NOTICE 'Администратор "%s" успешно добавлен.', admin\_name;

END;

$$ LANGUAGE plpgsql;

Листинг Б.8 – процедура delete\_admin

CREATE OR REPLACE PROCEDURE add\_rating\_by\_title(

film\_title VARCHAR(255),

user\_name VARCHAR(255),

user\_rating FLOAT

)

AS $$

DECLARE

film\_id INTEGER;

user\_id INTEGER;

BEGIN

-- Получаем идентификатор фильма по его названию

SELECT id\_film INTO film\_id FROM info\_films WHERE title = film\_title;

-- Получаем идентификатор пользователя по его имени

SELECT id\_customer INTO user\_id FROM customers WHERE name = user\_name;

-- Проверяем, существует ли пользователь и фильм

IF film\_id IS NULL THEN

RAISE EXCEPTION 'Фильм с названием "%", указанным в параметрах, не найден в базе данных.', film\_title;

END IF;

IF user\_id IS NULL THEN

RAISE EXCEPTION 'Пользователь с именем "%", указанным в параметрах, не найден в базе данных.', user\_name;

END IF;

-- Проверяем, существует ли уже рейтинг для этого фильма от этого пользователя

IF EXISTS (SELECT 1 FROM ratings WHERE id\_film = film\_id AND id\_customer = user\_id) THEN

RAISE EXCEPTION 'Рейтинг для фильма "%", уже существует от пользователя "%".', film\_title, user\_name;

END IF;

-- Добавляем рейтинг в таблицу ratings

INSERT INTO ratings (id\_film, id\_customer, rating) VALUES (film\_id, user\_id, user\_rating);

RAISE NOTICE 'Рейтинг для фильма "%", добавлен успешно.', film\_title;

EXCEPTION

WHEN OTHERS THEN

RAISE NOTICE 'Ошибка: %', SQLERRM;

END;

$$ LANGUAGE plpgsql;

Листинг Б.9 – процедура add\_rating\_by\_title

CREATE OR REPLACE PROCEDURE buy\_ticket(

p\_movie\_title VARCHAR(255),

p\_theater\_name VARCHAR(255),

p\_hall\_name VARCHAR(255),

p\_date DATE,

p\_time TIME,

p\_row INTEGER,

p\_seat INTEGER,

p\_customer\_name VARCHAR(255)

)

AS $$

DECLARE

movie\_id INTEGER;

theater\_id INTEGER;

hall\_id INTEGER;

timetable\_id INTEGER;

ticket\_price NUMERIC(10, 2);

customer\_id INTEGER;

order\_id INTEGER;

BEGIN

-- Проверка доступности билета

SELECT t.id\_timetable, tk.price

INTO timetable\_id, ticket\_price

FROM timetable t

JOIN halls h ON t.id\_hall = h.id\_hall

JOIN theaters th ON h.id\_theater = th.id\_theater

JOIN info\_films f ON t.id\_film = f.id\_film

JOIN tickets tk ON tk.id\_timetable = t.id\_timetable

WHERE f.title = p\_movie\_title

AND th.name = p\_theater\_name

AND h.name = p\_hall\_name

AND t.details->>'date' = to\_char(p\_date, 'YYYY-MM-DD')

AND t.details->>'time' = to\_char(p\_time, 'HH24:MI')

AND tk.id\_row = p\_row

AND tk.id\_seat = p\_seat

AND tk.status = 1;

IF NOT FOUND THEN

RAISE EXCEPTION 'Билет на указанный фильм, кинотеатр, зал, дату, время, ряд и место недоступен или не существует.';

END IF;

-- Получение ID покупателя

SELECT id\_customer INTO customer\_id FROM customers WHERE name = p\_customer\_name;

IF NOT FOUND THEN

RAISE EXCEPTION 'Покупатель с именем "%", указанным в параметрах, не найден в базе данных.', p\_customer\_name;

END IF;

-- Создание заказа

INSERT INTO orders (ord\_date, id\_customer)

VALUES (CURRENT\_DATE, customer\_id)

RETURNING id\_ord INTO order\_id;

-- Создание записи о купленном билете

INSERT INTO ord\_det (id\_ord, id\_row, id\_seat, id\_timetable)

VALUES (order\_id, p\_row, p\_seat, timetable\_id);

-- Обновление статуса билета на "куплен"

UPDATE tickets

SET status = 0

WHERE id\_row = p\_row

AND id\_seat = p\_seat

AND id\_timetable = timetable\_id;

-- Вывод сообщения о стоимости билета

RAISE NOTICE 'Билет куплен. Стоимость: %', ticket\_price;

END;

$$ LANGUAGE plpgsql;

Листинг Б.10 – процедура buy\_ticket

CREATE OR REPLACE PROCEDURE return\_ticket(

p\_movie\_title VARCHAR(255),

p\_theater\_name VARCHAR(255),

p\_hall\_name VARCHAR(255),

p\_date DATE,

p\_time TIME,

p\_row INTEGER,

p\_seat INTEGER,

p\_customer\_name VARCHAR(255)

)

security definer

AS $$

DECLARE

\_customer\_id INTEGER;

\_ticket\_id\_row INTEGER;

\_ticket\_id\_seat INTEGER;

\_ticket\_id\_timetable INTEGER;

\_ticket\_price NUMERIC(10, 2);

\_order\_id INTEGER;

BEGIN

-- Получаем ID покупателя

SELECT id\_customer INTO \_customer\_id FROM customers WHERE name = p\_customer\_name;

IF NOT FOUND THEN

RAISE EXCEPTION 'Покупатель с именем "%" не найден в базе данных.', p\_customer\_name;

END IF;

-- Получаем информацию о билете

SELECT tk.id\_row, tk.id\_seat, tk.id\_timetable, tk.price, o.id\_ord

INTO \_ticket\_id\_row, \_ticket\_id\_seat, \_ticket\_id\_timetable, \_ticket\_price, \_order\_id

FROM tickets tk

JOIN ord\_det od ON tk.id\_row = od.id\_row AND tk.id\_seat = od.id\_seat AND tk.id\_timetable = od.id\_timetable

JOIN orders o ON od.id\_ord = o.id\_ord

JOIN timetable t ON tk.id\_timetable = t.id\_timetable

JOIN halls h ON t.id\_hall = h.id\_hall

JOIN theaters th ON h.id\_theater = th.id\_theater

JOIN info\_films f ON t.id\_film = f.id\_film

WHERE f.title = p\_movie\_title

AND th.name = p\_theater\_name

AND h.name = p\_hall\_name

AND t.details->>'date' = to\_char(p\_date, 'YYYY-MM-DD')

AND t.details->>'time' = to\_char(p\_time, 'HH24:MI')

AND tk.id\_row = p\_row

AND tk.id\_seat = p\_seat;

IF NOT FOUND THEN

RAISE EXCEPTION 'Билет на указанный фильм, кинотеатр, зал, дату, время, ряд и место недоступен или не существует.';

END IF;

-- Проверяем, был ли билет куплен данным пользователем

IF NOT EXISTS (

SELECT 1

FROM ord\_det od

JOIN orders o ON od.id\_ord = o.id\_ord

WHERE od.id\_row = \_ticket\_id\_row

AND od.id\_seat = \_ticket\_id\_seat

AND od.id\_timetable = \_ticket\_id\_timetable

AND o.id\_customer = \_customer\_id

) THEN

RAISE EXCEPTION 'Билет не может быть возвращен, так как он зарегестрирован не на Вас.';

END IF;

-- Удаляем запись о билете из таблицы ord\_det

DELETE FROM ord\_det

WHERE id\_row = \_ticket\_id\_row

AND id\_seat = \_ticket\_id\_seat

AND id\_timetable = \_ticket\_id\_timetable;

-- Обновляем статус билета на доступный

UPDATE tickets

SET status = 1

WHERE id\_row = \_ticket\_id\_row

AND id\_seat = \_ticket\_id\_seat

AND id\_timetable = \_ticket\_id\_timetable;

-- Удаляем заказ

DELETE FROM orders WHERE id\_ord = \_order\_id;

-- Выводим сообщение о стоимости билета

RAISE NOTICE 'Билет успешно возвращен. Вам возвращено % рублей.', \_ticket\_price;

END;

$$ LANGUAGE plpgsql;

Листинг Б.11 – процедура return\_ticket

CREATE OR REPLACE PROCEDURE cancel\_ticket(

p\_movie\_title VARCHAR(255),

p\_theater\_name VARCHAR(255),

p\_hall\_name VARCHAR(255),

p\_date DATE,

p\_time TIME,

p\_row INTEGER,

p\_seat INTEGER

)

security definer

AS $$

DECLARE

movie\_id INTEGER;

theater\_id INTEGER;

hall\_id INTEGER;

timetable\_id INTEGER;

ticket\_price NUMERIC(10, 2);

BEGIN

-- Проверка доступности билета

SELECT t.id\_timetable, tk.price

INTO timetable\_id, ticket\_price

FROM timetable t

JOIN halls h ON t.id\_hall = h.id\_hall

JOIN theaters th ON h.id\_theater = th.id\_theater

JOIN info\_films f ON t.id\_film = f.id\_film

JOIN tickets tk ON tk.id\_timetable = t.id\_timetable

WHERE f.title = p\_movie\_title

AND th.name = p\_theater\_name

AND h.name = p\_hall\_name

AND t.details->>'date' = to\_char(p\_date, 'YYYY-MM-DD')

AND t.details->>'time' = to\_char(p\_time, 'HH24:MI')

AND tk.id\_row = p\_row

AND tk.id\_seat = p\_seat

AND tk.status = 1;

IF NOT FOUND THEN

RAISE EXCEPTION 'Билет на указанный фильм, кинотеатр, зал, дату, время, ряд и место недоступен или не существует.';

END IF;

-- Обновление статуса билета на "куплен"

UPDATE tickets

SET status = 0

WHERE id\_row = p\_row

AND id\_seat = p\_seat

AND id\_timetable = timetable\_id;

-- Вывод сообщения о стоимости билета

RAISE NOTICE 'Билет удален из доступа.';

END;

$$ LANGUAGE plpgsql;

Листинг Б.12 – процедура cancel\_ticket

CREATE OR REPLACE PROCEDURE ExportCountTicketsFromTheaterToXML(

file\_path TEXT

)

security definer

AS $$

DECLARE

xml\_data TEXT := '<?xml version="1.0" encoding="UTF-8"?><TheatersTickets>';

theater\_movie\_ticket\_rec RECORD;

BEGIN

FOR theater\_movie\_ticket\_rec IN SELECT \* FROM temp\_theater\_movie\_tickets LOOP

xml\_data := xml\_data || '<theater\_movie\_ticket>';

xml\_data := xml\_data || '<theater\_name>' || theater\_movie\_ticket\_rec.theater\_name || '</theater\_name>';

xml\_data := xml\_data || '<movie\_title>' || theater\_movie\_ticket\_rec.movie\_title || '</movie\_title>';

xml\_data := xml\_data || '<total\_tickets\_sold>' || theater\_movie\_ticket\_rec.total\_tickets\_sold || '</total\_tickets\_sold>';

xml\_data := xml\_data || '<from\_date>' || theater\_movie\_ticket\_rec.first\_purchase\_date || '</from\_date>';

xml\_data := xml\_data || '<to\_date>' || theater\_movie\_ticket\_rec.last\_purchase\_date || '</to\_date>';

xml\_data := xml\_data || '</theater\_movie\_ticket>';

END LOOP;

xml\_data := xml\_data || '</TheaterTickets>';

EXECUTE format('COPY (SELECT %L) TO %L', xml\_data, file\_path);

RAISE NOTICE 'Данные успешно загружены в XML файл: %', file\_path;

EXCEPTION

WHEN OTHERS THEN

RAISE NOTICE 'Ошибка при экспортировке данных в XML: %', SQLERRM;

END;

$$ LANGUAGE plpgsql;

Листинг Б.13 – процедура ExportCountTicketsFromTheaterToXML

CREATE OR REPLACE PROCEDURE ImportFilmsFromXML(file\_path VARCHAR)

security definer

AS $$

DECLARE

xml\_data TEXT;

BEGIN

-- Read data from the file

xml\_data := pg\_read\_file(file\_path);

-- Check if data is read successfully

IF xml\_data IS NULL THEN

RAISE EXCEPTION 'Failed to read data from file %', file\_path;

END IF;

-- Display the data read from the file for debugging

RAISE INFO 'Данные проверяются из файла: %', xml\_data;

-- Create a temporary table for importing data

CREATE TEMP TABLE tmp\_films (

title VARCHAR(255),

genre VARCHAR(255),

rating INT,

description VARCHAR(2000)

);

-- Insert new data from XML into the temporary table

BEGIN

EXECUTE 'INSERT INTO tmp\_films (title, genre, rating, description)

SELECT

unnest(xpath(''/info\_films/info\_films/title/text()'',

xmlparse(document ''' || xml\_data || ''')))::text AS title,

unnest(xpath(''/info\_films/info\_films/genre/text()'',

xmlparse(document ''' || xml\_data || ''')))::text AS genre,

CAST(TRIM(unnest(xpath(''/info\_films/info\_films/rating/text()'',

xmlparse(document ''' || xml\_data || ''')))::text) AS INT) AS rating,

unnest(xpath(''/info\_films/info\_films/description/text()'',

xmlparse(document ''' || xml\_data || ''')))::text AS description';

EXCEPTION

WHEN OTHERS THEN

RAISE EXCEPTION 'Возникла ошибка с импортом данных из XML: %', SQLERRM;

END;

-- Insert data from the temporary table into info\_films

INSERT INTO info\_films (title, genre, rating, description)

SELECT title, genre, rating, description FROM tmp\_films;

RAISE INFO 'Данный каталог был успешно перенесен из файла % во временную таблицу tmp\_films', file\_path;

END;

$$ LANGUAGE plpgsql;

Листинг Б.14 – процедура ImportFilmsFromXML

# **Приложение В**

Листинг функций

CREATE OR REPLACE FUNCTION generate\_tickets()

RETURNS TRIGGER AS $$

DECLARE

hall\_row\_count INTEGER;

hall\_seat\_count INTEGER;

curr\_row INTEGER;

curr\_seat INTEGER;

ticket\_price NUMERIC(10, 2);

hall\_coefficient NUMERIC(10, 2);

timetable\_coefficient NUMERIC(10, 2);

BEGIN

-- Получение количества рядов и мест в зале

SELECT rows, seats, coefficient INTO hall\_row\_count, hall\_seat\_count, hall\_coefficient

FROM halls

WHERE id\_hall = (

SELECT id\_hall FROM timetable WHERE id\_timetable = NEW.id\_timetable

);

curr\_row := 1;

curr\_seat := 1;

-- Получение коэффициента из деталей расписания

SELECT (NEW.details->>'coefficient')::NUMERIC INTO timetable\_coefficient;

-- Генерация записей в таблицу "tickets" для каждого ряда и места

WHILE curr\_row <= hall\_row\_count LOOP

WHILE curr\_seat <= hall\_seat\_count LOOP

-- Получение базовой цены из таблицы halls

SELECT def\_price INTO ticket\_price

FROM halls

WHERE id\_hall = (

SELECT id\_hall FROM timetable WHERE id\_timetable = NEW.id\_timetable

);

-- Проверка совпадения с bestrows и bestseats

IF curr\_row = ANY(ARRAY(SELECT jsonb\_array\_elements\_text(bestrows)::INTEGER FROM halls WHERE id\_hall = NEW.id\_hall)::INTEGER[]) AND

curr\_seat = ANY(ARRAY(SELECT jsonb\_array\_elements\_text(bestseats)::INTEGER FROM halls WHERE id\_hall = NEW.id\_hall)::INTEGER[]) THEN

ticket\_price := ticket\_price \* hall\_coefficient \* timetable\_coefficient;

ELSE

ticket\_price := ticket\_price \* timetable\_coefficient;

END IF;

-- Вставка записи в таблицу "tickets"

INSERT INTO tickets (id\_row, id\_seat, id\_timetable, price, status)

VALUES (curr\_row, curr\_seat, NEW.id\_timetable, ticket\_price, 1);

curr\_seat := curr\_seat + 1; -- Переход к следующему месту

END LOOP;

curr\_row := curr\_row + 1; -- Переход к следующему ряду

curr\_seat := 1; -- Сброс счетчика мест

END LOOP;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

Листинг В.1 – функция generate\_tickets

CREATE OR REPLACE FUNCTION delete\_related\_tickets()

RETURNS TRIGGER AS $$

BEGIN

-- Удаление билетов, связанных с удаляемой записью из таблицы "timetable"

DELETE FROM tickets

WHERE id\_timetable = OLD.id\_timetable;

RETURN OLD;

END;

$$ LANGUAGE plpgsql;

Листинг В.2 – функция delete\_related\_tickets

CREATE OR REPLACE FUNCTION update\_tickets\_on\_timetable\_update()

RETURNS TRIGGER AS $$

DECLARE

hall\_row\_count INTEGER;

hall\_seat\_count INTEGER;

curr\_row INTEGER;

curr\_seat INTEGER;

ticket\_price NUMERIC(10, 2);

hall\_coefficient NUMERIC(10, 2);

timetable\_coefficient NUMERIC(10, 2);

BEGIN

-- Получение количества рядов и мест в зале

SELECT rows, seats, coefficient INTO hall\_row\_count, hall\_seat\_count, hall\_coefficient

FROM halls

WHERE id\_hall = (

SELECT id\_hall FROM timetable WHERE id\_timetable = NEW.id\_timetable

);

curr\_row := 1;

curr\_seat := 1;

-- Получение коэффициента из деталей расписания

SELECT (NEW.details->>'coefficient')::NUMERIC INTO timetable\_coefficient;

-- Обновление данных в таблице "tickets" для каждого ряда и места

WHILE curr\_row <= hall\_row\_count LOOP

WHILE curr\_seat <= hall\_seat\_count LOOP

-- Получение базовой цены из таблицы halls

SELECT def\_price INTO ticket\_price

FROM halls

WHERE id\_hall = (

SELECT id\_hall FROM timetable WHERE id\_timetable = NEW.id\_timetable

);

-- Проверка совпадения с bestrows и bestseats

IF curr\_row = ANY(ARRAY(SELECT jsonb\_array\_elements\_text(bestrows)::INTEGER FROM halls WHERE id\_hall = NEW.id\_hall)::INTEGER[]) AND

curr\_seat = ANY(ARRAY(SELECT jsonb\_array\_elements\_text(bestseats)::INTEGER FROM halls WHERE id\_hall = NEW.id\_hall)::INTEGER[]) THEN

ticket\_price := ticket\_price \* hall\_coefficient \* timetable\_coefficient;

ELSE

ticket\_price := ticket\_price \* timetable\_coefficient;

END IF;

-- Обновление записи в таблице "tickets"

UPDATE tickets

SET price = ticket\_price,

status = CASE WHEN NEW.status = 'В прокате' THEN 1 ELSE 0 END

WHERE id\_row = curr\_row

AND id\_seat = curr\_seat

AND id\_timetable = NEW.id\_timetable;

curr\_seat := curr\_seat + 1; -- Переход к следующему месту

END LOOP;

curr\_row := curr\_row + 1; -- Переход к следующему ряду

curr\_seat := 1; -- Сброс счетчика мест

END LOOP;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

Листинг В.3 – функция update\_tickets\_on\_timetable\_update

CREATE OR REPLACE FUNCTION search\_tickets\_by\_movie\_title(movie\_title\_param VARCHAR(255))

RETURNS TABLE (

movie\_title VARCHAR(255),

hall\_name VARCHAR(255),

session\_date JSONB,

session\_time JSONB,

available\_tickets BIGINT

)

security definer

AS $$

DECLARE

movie\_found BOOLEAN;

BEGIN

movie\_found := FALSE;

FOR movie\_title, hall\_name, session\_date, session\_time, available\_tickets IN

SELECT i.title AS movie\_title,

h.name AS hall\_name,

TO\_JSONB(t.details->>'date') AS session\_date,

TO\_JSONB(t.details->>'time') AS session\_time,

COUNT(tk.id\_timetable) AS available\_tickets

FROM info\_films i

INNER JOIN timetable t ON i.id\_film = t.id\_film

INNER JOIN halls h ON t.id\_hall = h.id\_hall

LEFT JOIN tickets tk ON t.id\_timetable = tk.id\_timetable AND tk.status = 1

WHERE i.title = movie\_title\_param

GROUP BY i.title, h.name, t.details

ORDER BY h.name

LOOP

movie\_found := TRUE;

RETURN NEXT;

END LOOP;

IF NOT movie\_found THEN

RAISE NOTICE 'Фильм с названием "%" не найден.', movie\_title\_param;

END IF;

RETURN;

END;

$$ LANGUAGE plpgsql;

Листинг В.4 – функция search\_tickets\_by\_movie\_title

CREATE OR REPLACE FUNCTION search\_tickets\_by\_session\_date(session\_date\_param DATE)

RETURNS TABLE (

movie\_title VARCHAR(255),

hall\_name VARCHAR(255),

session\_date JSONB,

session\_time JSONB,

available\_tickets BIGINT

)

security definer

AS $$

DECLARE

movies\_found BOOLEAN;

BEGIN

movies\_found := FALSE;

FOR movie\_title, hall\_name, session\_date, session\_time, available\_tickets IN

SELECT i.title AS movie\_title,

h.name AS hall\_name,

TO\_JSONB(t.details->>'date') AS session\_date,

TO\_JSONB(t.details->>'time') AS session\_time,

COUNT(tk.id\_timetable) AS available\_tickets

FROM info\_films i

INNER JOIN timetable t ON i.id\_film = t.id\_film

INNER JOIN halls h ON t.id\_hall = h.id\_hall

LEFT JOIN tickets tk ON t.id\_timetable = tk.id\_timetable AND tk.status = 1

WHERE t.details->>'date' = TO\_CHAR(session\_date\_param, 'YYYY-MM-DD')

GROUP BY i.title, h.name, t.details

ORDER BY h.name

LOOP

movies\_found := TRUE;

RETURN NEXT;

END LOOP;

IF NOT movies\_found THEN

RAISE NOTICE 'На % дату нет фильмов.', TO\_CHAR(session\_date\_param, 'YYYY-MM-DD');

END IF;

RETURN;

END;

$$ LANGUAGE plpgsql;

Листинг В.5 – функция search\_tickets\_by\_session\_date

CREATE OR REPLACE FUNCTION search\_tickets\_by\_genre(genre\_param VARCHAR(255))

RETURNS TABLE (

movie\_title VARCHAR(255),

hall\_name VARCHAR(255),

session\_date JSONB,

session\_time JSONB,

available\_tickets BIGINT

)

security definer

AS $$

DECLARE

movies\_found BOOLEAN;

BEGIN

movies\_found := FALSE;

FOR movie\_title, hall\_name, session\_date, session\_time, available\_tickets IN

SELECT i.title AS movie\_title,

h.name AS hall\_name,

TO\_JSONB(t.details->>'date') AS session\_date,

TO\_JSONB(t.details->>'time') AS session\_time,

COUNT(tk.id\_timetable) AS available\_tickets

FROM info\_films i

INNER JOIN timetable t ON i.id\_film = t.id\_film

INNER JOIN halls h ON t.id\_hall = h.id\_hall

LEFT JOIN tickets tk ON t.id\_timetable = tk.id\_timetable AND tk.status = 1

WHERE i.genre = genre\_params

GROUP BY i.title, h.name, t.details

ORDER BY h.name

LOOP

movies\_found := TRUE;

RETURN NEXT;

END LOOP;

IF NOT movies\_found THEN

RAISE NOTICE 'Фильмов жанра % не найдено.', genre\_param;

END IF;

RETURN;

END;

$$ LANGUAGE plpgsql;

Листинг В.6 – функция search\_tickets\_by\_genre

CREATE OR REPLACE FUNCTION sort\_films\_by\_rating()

RETURNS TABLE (

movie\_title VARCHAR(255),

genre VARCHAR(255),

rating FLOAT,

start\_date DATE,

end\_date DATE

)

security definer

AS $$

BEGIN

RETURN QUERY

WITH timetable\_dates AS (

SELECT id\_film,

MIN((details->>'date')::DATE) AS start\_date,

MAX((details->>'date')::DATE) AS end\_date

FROM timetable

GROUP BY id\_film

)

SELECT i.title AS movie\_title,

i.genre,

i.rating,

td.start\_date,

td.end\_date

FROM info\_films i

INNER JOIN timetable\_dates td ON i.id\_film = td.id\_film

ORDER BY i.rating DESC;

END;

$$ LANGUAGE plpgsql;

Листинг В.7 – функция sort\_films\_by\_rating

CREATE OR REPLACE FUNCTION find\_tickets\_by\_quantity(num\_tickets\_needed INTEGER)

RETURNS TABLE (

movie\_title VARCHAR(255),

film\_genre VARCHAR(255),

film\_rating FLOAT,

day DATE,

show\_time TIME,

available\_tickets BIGINT

)

security definer

AS $$

BEGIN

RETURN QUERY

SELECT

f.title AS movie\_title,

f.genre AS film\_genre,

f.rating AS film\_rating,

(t.details->>'date')::DATE AS day,

(t.details->>'time')::TIME AS show\_time,

COUNT(\*) AS available\_tickets

FROM timetable t

INNER JOIN info\_films f ON t.id\_film = f.id\_film

LEFT JOIN tickets ti ON t.id\_timetable = ti.id\_timetable

AND ti.status = 1

GROUP BY f.title, f.genre, f.rating, (t.details->>'date')::DATE, (t.details->>'time')::TIME

HAVING COUNT(\*) >= num\_tickets\_needed;

END;

$$ LANGUAGE plpgsql;

Листинг В.8 – функция find\_tickets\_by\_quantity

CREATE OR REPLACE FUNCTION get\_available\_tickets\_summary()

RETURNS TABLE (

movie\_title VARCHAR(255),

film\_genre VARCHAR(255),

film\_rating FLOAT,

theater\_name VARCHAR(255),

film\_status VARCHAR(20),

total\_available\_tickets BIGINT

)

security definer

AS $$

BEGIN

RETURN QUERY

SELECT

f.title AS movie\_title,

f.genre AS film\_genre,

f.rating AS film\_rating,

th.name AS theater\_name,

CASE

WHEN EXISTS (SELECT 1 FROM timetable t WHERE t.id\_film = f.id\_film) THEN 'В прокате'::VARCHAR(20)

ELSE 'Ожидается в прокате'::VARCHAR(20)

END AS film\_status,

COALESCE(SUM(CASE WHEN ti.status = 1 THEN 1 ELSE 0 END), 0) AS total\_available\_tickets

FROM info\_films f

LEFT JOIN timetable t ON f.id\_film = t.id\_film

LEFT JOIN halls h ON t.id\_hall = h.id\_hall

LEFT JOIN theaters th ON h.id\_theater = th.id\_theater

LEFT JOIN tickets ti ON t.id\_timetable = ti.id\_timetable

GROUP BY f.id\_film, th.name;

END;

$$ LANGUAGE plpgsql;

Листинг В.9 – функция get\_available\_tickets\_summary

CREATE OR REPLACE FUNCTION get\_tickets\_info()

RETURNS TABLE (

movie\_title VARCHAR(255),

film\_genre VARCHAR(255),

film\_rating FLOAT,

available\_tickets BIGINT,

ticket\_price\_range TEXT

)

security definer

AS $$

BEGIN

RETURN QUERY

SELECT

f.title AS movie\_title,

f.genre AS film\_genre,

f.rating AS film\_rating,

COUNT(t.id\_row)::BIGINT AS available\_tickets,

(MIN(t.price)::VARCHAR(10) || ' - ' || MAX(t.price)::VARCHAR(10)) AS ticket\_price\_range

FROM

info\_films f

LEFT JOIN

timetable tt ON f.id\_film = tt.id\_film

LEFT JOIN

tickets t ON tt.id\_timetable = t.id\_timetable AND t.status = 1

WHERE

tt.id\_film IS NOT NULL

GROUP BY

f.title, f.genre, f.rating;

END;

$$ LANGUAGE plpgsql;

Листинг В.10 – функция get\_tickets\_info

CREATE OR REPLACE FUNCTION available\_movie\_tickets\_price(price\_range NUMERIC) RETURNS TABLE (

film\_title VARCHAR(255),

film\_rating FLOAT,

film\_genre VARCHAR(255),

available\_tickets BIGINT,

ticket\_price\_range JSONB

)

security definer

AS $$

BEGIN

RETURN QUERY

SELECT

f.title AS film\_title,

f.rating AS film\_rating,

f.genre AS film\_genre,

COUNT(t.id\_row) AS available\_tickets,

jsonb\_build\_object(

'min\_price', MIN(t.price),

'max\_price', MAX(t.price)

) AS ticket\_price\_range

FROM

info\_films f

JOIN

timetable tt ON f.id\_film = tt.id\_film

JOIN

tickets t ON tt.id\_timetable = t.id\_timetable

WHERE

t.status = 1

GROUP BY

f.title, f.rating, f.genre

HAVING

COUNT(t.id\_row) > 0 -- Считаем количество строк по рандомному полю

AND MIN(t.price) <= price\_range AND MAX(t.price) >= price\_range;

END;

$$ LANGUAGE plpgsql;

Листинг В.11 – функция available\_movie\_tickets\_price

CREATE OR REPLACE FUNCTION update\_average\_rating()

RETURNS TRIGGER AS $$

BEGIN

UPDATE info\_films AS f

SET rating = (

SELECT AVG(rating)

FROM ratings

WHERE id\_film = NEW.id\_film

)

WHERE f.id\_film = NEW.id\_film;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

Листинг В.12 – функция update\_average\_rating

CREATE OR REPLACE FUNCTION add\_customer(

p\_customer\_name VARCHAR(255)

)

RETURNS VOID

security definer

AS $$

DECLARE

\_existing\_customer BOOLEAN;

BEGIN

-- Проверяем наличие покупателя с таким именем

SELECT EXISTS(SELECT 1 FROM customers WHERE name = p\_customer\_name) INTO \_existing\_customer;

-- Если покупатель уже существует, выводим сообщение

IF \_existing\_customer THEN

RAISE NOTICE 'Покупатель "%" уже существует.', p\_customer\_name;

ELSE

-- Вставляем данные о покупателе в таблицу customers

INSERT INTO customers (name) VALUES (p\_customer\_name);

-- Выводим сообщение об успешном добавлении

RAISE NOTICE 'Данные о покупателе "%" успешно добавлены в таблицу customers.', p\_customer\_name;

END IF;

END;

$$ LANGUAGE plpgsql;

Листинг В.13 – функция add\_customer

CREATE OR REPLACE FUNCTION get\_user\_ticket\_purchases(user\_name VARCHAR(255))

RETURNS TABLE (

user\_email VARCHAR(255),

purchase\_date TIMESTAMP without time zone,

total\_ticket\_price NUMERIC

)

security definer

AS $$

BEGIN

RETURN QUERY

SELECT

user\_ticket\_purchases.user\_email,

user\_ticket\_purchases.purchase\_date,

user\_ticket\_purchases.total\_ticket\_price

FROM

user\_ticket\_purchases

WHERE

user\_ticket\_purchases.user\_email = user\_name;

END;

$$ LANGUAGE plpgsql;

Листинг В.14 – функция get\_user\_ticket\_purchases

CREATE OR REPLACE FUNCTION get\_tickets\_sold\_per\_theater()

RETURNS TABLE (

theater\_name VARCHAR(255),

total\_tickets\_sold INTEGER

)

security definer

AS $$

BEGIN

RETURN QUERY

SELECT

th.name AS theater\_name,

COUNT(\*)::INTEGER AS total\_tickets\_sold

FROM

orders o

JOIN

ord\_det od ON o.id\_ord = od.id\_ord

JOIN

tickets t ON od.id\_row = t.id\_row AND od.id\_seat = t.id\_seat AND od.id\_timetable = t.id\_timetable

JOIN

timetable tt ON t.id\_timetable = tt.id\_timetable

JOIN

halls h ON tt.id\_hall = h.id\_hall

JOIN

theaters th ON h.id\_theater = th.id\_theater

GROUP BY

th.name;

END;

$$ LANGUAGE plpgsql;

Листинг В.15 – функция get\_tickets\_sold\_per\_theater

CREATE OR REPLACE FUNCTION get\_theater\_movie\_tickets(theater\_name VARCHAR(255), from\_date DATE, to\_date DATE)

RETURNS VOID

SECURITY DEFINER

AS $$

BEGIN

-- Drop the temporary table if it already exists

IF EXISTS (SELECT 1 FROM pg\_tables WHERE tablename = 'temp\_theater\_movie\_tickets') THEN

DROP TABLE temp\_theater\_movie\_tickets;

END IF;

-- Create the temporary table

CREATE TEMP TABLE temp\_theater\_movie\_tickets AS

SELECT

tm.theater\_name,

tm.movie\_title,

tm.total\_tickets\_sold,

tm.first\_purchase\_date,

tm.last\_purchase\_date

FROM

theaters\_movie\_tickets tm

WHERE

tm.theater\_name = get\_theater\_movie\_tickets.theater\_name

AND tm.first\_purchase\_date >= get\_theater\_movie\_tickets.from\_date

AND tm.last\_purchase\_date <= get\_theater\_movie\_tickets.to\_date;

END;

$$ LANGUAGE plpgsql;

Листинг В.16 – функция get\_theater\_movie\_tickets