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

Факультет Информационных технологий

Кафедра Информационных систем и технологий

Специальность 1-40 05 01 «Информационные системы и технологии»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА КУРСОВОГО ПРОЕКТА**

по дисциплине «Базы данных»

Тема: «Реализация базы данных для музыкальной площадки с использованием технологии применения мультимедийных типов данных»

**Исполнитель**

студент 2 курса 2 группы \_\_\_\_\_\_\_\_\_\_\_\_\_\_ В. С. Петрикевич

подпись, дата

**Руководитель**

Ассистент Н. И. Уласевич

(должность, уч. звание) (подпись, дата)

Допущен(а) к защите \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_

дата, подпись

Курсовой проект защищен с оценкой

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Н. И. Уласевич

подпись дата инициалы и фамилия

**Содержание**

[Введение 5](#_Toc167088299)

[1 Анализ требований к программному средству 6](#_Toc167088300)

[1.1 Аналитический обзор аналогов 6](#_Toc167088301)

[1.1.1 Аналог «Spotify» 6](#_Toc167088302)

[1.1.2 Аналог «Apple Music» 7](#_Toc167088303)

[1.2 Разработка функциональных требований, определение вариантов использования 8](#_Toc167088304)

[1.3 Вывод 10](#_Toc167088305)

[2 Разработка объектов базы данных 11](#_Toc167088306)

[2.1 Таблицы 11](#_Toc167088307)

[2.2 Разработка представлений базы данных 14](#_Toc167088308)

[2.3 Разработка индексов базы данных 15](#_Toc167088309)

[2.4 Триггеры базы данных 16](#_Toc167088310)

[2.5 Разработка функций базы данных 17](#_Toc167088311)

[2.5.1 Выборка данных из таблиц 17](#_Toc167088312)

[2.5.2 Выборка данных по поисковому запросу 18](#_Toc167088313)

[2.5.3 Заполнение таблиц 100 000 строк 20](#_Toc167088314)

[2.5.4 Добавление данных в таблицы 20](#_Toc167088315)

[25.5 Удаление данных в таблицы 22](#_Toc167088316)

[2.5.6 Изменение данных в таблицы 22](#_Toc167088317)

[2.5.7 Дополнительные функции 23](#_Toc167088318)

[2.6 Разработка триггеров базы данных 25](#_Toc167088319)

[2.7 Разработка пользователей 26](#_Toc167088320)

[2.8 Вывод 27](#_Toc167088321)

[3 Описание процедур импорта и экспорта 28](#_Toc167088322)

[4 Тестирование производительности 31](#_Toc167088323)

[4.2 Вывод 32](#_Toc167088324)

[5 Описание технологии и ее применение в базе данных 33](#_Toc167088325)

[5.1 Применение технологии хранения мультимедийных типов данных в базе данных 33](#_Toc167088326)

[5.2 Второй способ хранения мультимедийных типов данных в базе данных 33](#_Toc167088327)

[5.3 Вывод 34](#_Toc167088328)

[6 Руководство пользователя 35](#_Toc167088329)

[Заключение 36](#_Toc167088330)

[Список используемых источников 37](#_Toc167088331)

[Приложение А. Код создания таблиц 38](#_Toc167088332)

[Приложение Б. Код создания функций 40](#_Toc167088333)

[Приложение В. Выполнение процедур 43](#_Toc167088334)

[Приложение Д. Роли клиентов 45](#_Toc167088335)

# Введение

Цель данной работы заключается в создании реляционной базы данных для музыкальной платформы, которая обеспечивает пользователя доступом к имеющимся на ней музыкальным композициям.

База данных - организованное собрание данных, которое обычно хранится в электронном виде в компьютерной системе. БД используются для хранения, организации и управления большим объемом структурированных и неструктурированных данных. Реляционная база данных является наиболее распространенной формой организации данных, в которой данные представлены в виде таблиц, состоящих из строк и столбцов, где каждый столбец представляет атрибут, а каждая строка представляет кортеж или запись. В данной работе для управления базой данных была выбрана СУБД Postgres SQL, поскольку эта система обладает высокой надежностью и производительностью, что позволяет обеспечить эффективное хранение, обработку и управление музыкальными данными.

Основные требования к базе данных:

* Реализация ролей администратора и пользователя;
* поиск аудиозаписей по альбому, исполнителю или жанру;
* загрузка аудиозаписей на платформу пользователем;
* взаимодействие с базой данных при помощи хранимых процедур и функций.

В пояснительной записке содержится информация о сопоставимых продуктах, структуре и реализации проекта.

**1 Анализ требований к программному средству**

1. **Аналитический обзор аналогов**

Музыка - одна из самых популярных и распространенных форм искусства, которая имеет давнюю историю и оказывает значительное влияние на культуру и общество в целом. В настоящее время музыкальные платформы являются важной частью музыкальной индустрии, обеспечивая людям доступ к огромной библиотеке музыкальных произведений различных жанров и эпох.

Одним из основных преимуществ музыкальных платформ является возможность слушать музыку в любое время и в любом месте с помощью смартфона, компьютера или другого устройства с доступом в Интернет. Это делает музыкальные платформы очень популярными среди миллионов пользователей по всему миру.

**1.1.1 Аналог «Spotify»**

Spotify - популярная музыкальная платформа, которая позволяет пользователям стримить музыку онлайн, создавать плейлисты и делиться ими с друзьями, слушать радио, а также находить новую музыку, основываясь на своих предпочтениях. Пример интерфейса данного сервиса представлен на рисунке 1.1.

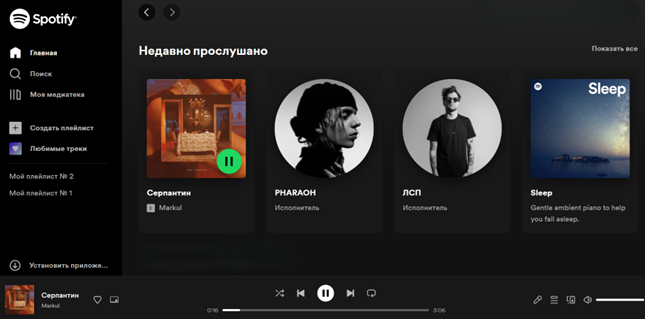


Рисунок 1.1 – Интерфейс сервиса Spotify

Одной из основных функций Spotify является поиск музыки. Пользователи могут искать музыку по альбому, исполнителю, жанру или песне. Кроме того, Spotify предлагает персонализированные рекомендации в соответствии с предпочтениями пользователя, а также плейлисты, созданные другими пользователями и кураторами платформы.

Пользователи могут создавать свои собственные плейлисты и делиться ими с друзьями или сохранять музыку для офлайн-воспроизведения. Spotify также позволяет пользователям слушать радио и подкасты, включая оригинальный контент, созданный самой платформой.

Другая функция Spotify - возможность использовать платформу как социальную сеть, где пользователи могут подписываться на друг друга, просматривать их плейлисты и рекомендации, а также обмениваться сообщениями.

Spotify также предлагает два варианта подписки: бесплатную и платную. Бесплатная версия содержит рекламу и ограничения в использовании, в то время как платная версия позволяет получить неограниченный доступ к музыке, отсутствие рекламы и другие функции.

Spotify использует несколько баз данных в своей архитектуре, включая Cassandra и Apache Kafka. Cassandra - это высокомасштабируемая NoSQL база данных, которая используется для хранения метаданных треков, включая информацию об альбомах, исполнителях, жанрах и т.д. Cassandra также используется для хранения информации о пользовательских аккаунтах и настроек. Apache Kafka - это распределенная платформа потоковой обработки данных, которая используется для передачи и обработки потоковых данных между различными компонентами Spotify, такими как служба рекомендаций, система поиска и т.д.

В общем, Spotify использует высокопроизводительные базы данных для обеспечения быстрого доступа к большому объему музыкальных данных и обработки потоковых данных для предоставления персонализированных рекомендаций и других функций платформы.

**1.1.2 Аналог «Apple Music»**

Apple Music - это популярная музыкальная платформа, которая позволяет пользователям стримить музыку онлайн. Apple Music позволяет пользователям прослушивать музыку на различных устройствах, а также создавать собственные плейлисты и подборки.

Apple Music предлагает пользователю возможность доступа к более чем 75 миллионам песен, а также к подкастам и аудиокнигам. Сервис также предлагает персонализированные рекомендации, основанные на предпочтениях пользователя.

Пользователи могут создавать свои плейлисты и делиться ими с другими пользователями. Apple Music также предлагает радиостанции, созданные известными диджеями и музыкантами, а также живые трансляции концертов и мероприятий.

Интерфейс Apple Music схож с интерфейсом Spotify и предлагает удобный поиск по альбомам, песням и исполнителям. Также сервис позволяет скачивать музыку для офлайн прослушивания. Пример интерфейса данного сервиса представлен на рисунке 1.2.

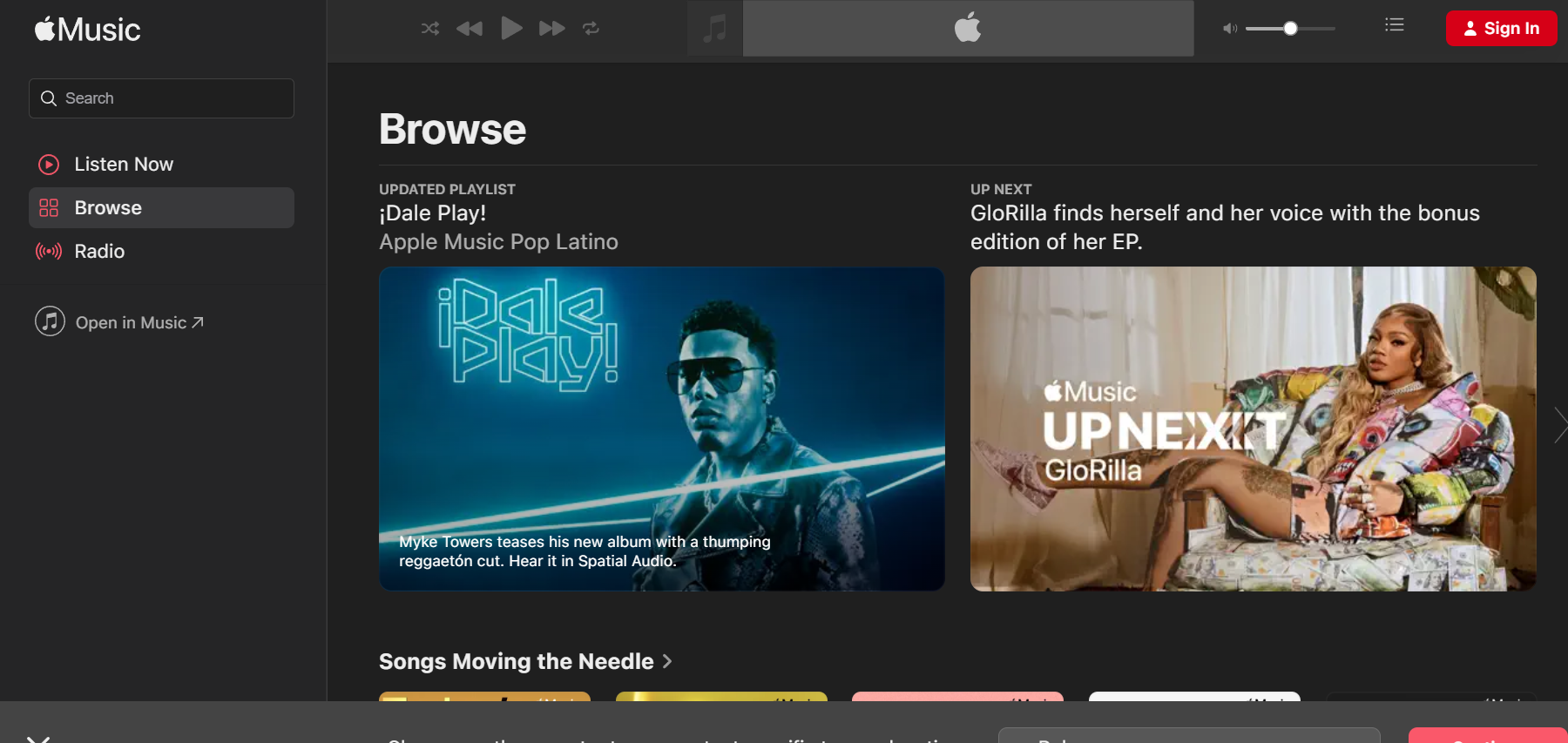


Рисунок 1.2 – Интерфейс сервиса Apple Music

Apple Music использует несколько баз данных в своей архитектуре, включая MySQL и Cassandra. MySQL - это реляционная база данных, которая используется для хранения пользовательской информации, такой как логины и пароли, а также информации о подписке и оплате. Cassandra также используется для хранения метаданных треков, включая информацию об альбомах, исполнителях, жанрах и т.д., а также для предоставления быстрого доступа к этой информации пользователям. Кроме того, Apple Music использует инфраструктуру Amazon Web Services для обеспечения высокой доступности и масштабируемости своей платформы.

В целом, Apple Music также использует масштабируемые и высокопроизводительные базы данных для обеспечения быстрого доступа к большому объему музыкальных данных и управления информацией о пользователях и их подписках.

**1.2 Разработка функциональных требований, определение вариантов использования**

Функциональные требования базы данных определяют, как база данных должна обрабатывать данные и предоставлять пользовательскому интерфейсу необходимую функциональность. Это может включать в себя описание того, как данные должны храниться и организовываться, как происходит поиск и выборка данных, каким образом обновляются данные и какие механизмы используются для защиты данных. Кроме того, функциональные требования могут определять интеграцию базы данных с другими системами и программами. Например, для музыкальной площадки функциональные требования могут включать в себя функции для хранения информации о музыкальных треках и пользователях, поиска музыки по категориям и критериям, создания и управления плейлистами, а также функции для оценки и прослушивания музыки.

Помимо функциональных требований, важно также определить роли пользователей и их варианты использования системы. Варианты использования описывают, как пользователи будут взаимодействовать с системой в зависимости от своих ролей. Это помогает определить, какие функции должны быть доступны для каждой роли, какие данные должны быть доступны для каждой роли, а также как должна быть организована навигация в системе. Варианты использования обычно представляются в виде UML диаграмм, которые позволяют наглядно отобразить взаимодействие между пользователями и системой.

Роли пользователя — это набор прав, которые пользователь может получить в системе. В зависимости от роли пользователя, он может иметь доступ к различным функциям системы. В данном проекте роли пользователей будут следующими:

* Guest.
* User.
* Manager.

На основе предоставленного списка ролей необходимо построить варианты использование. Варианты использование изображена на рисунке 1.3.

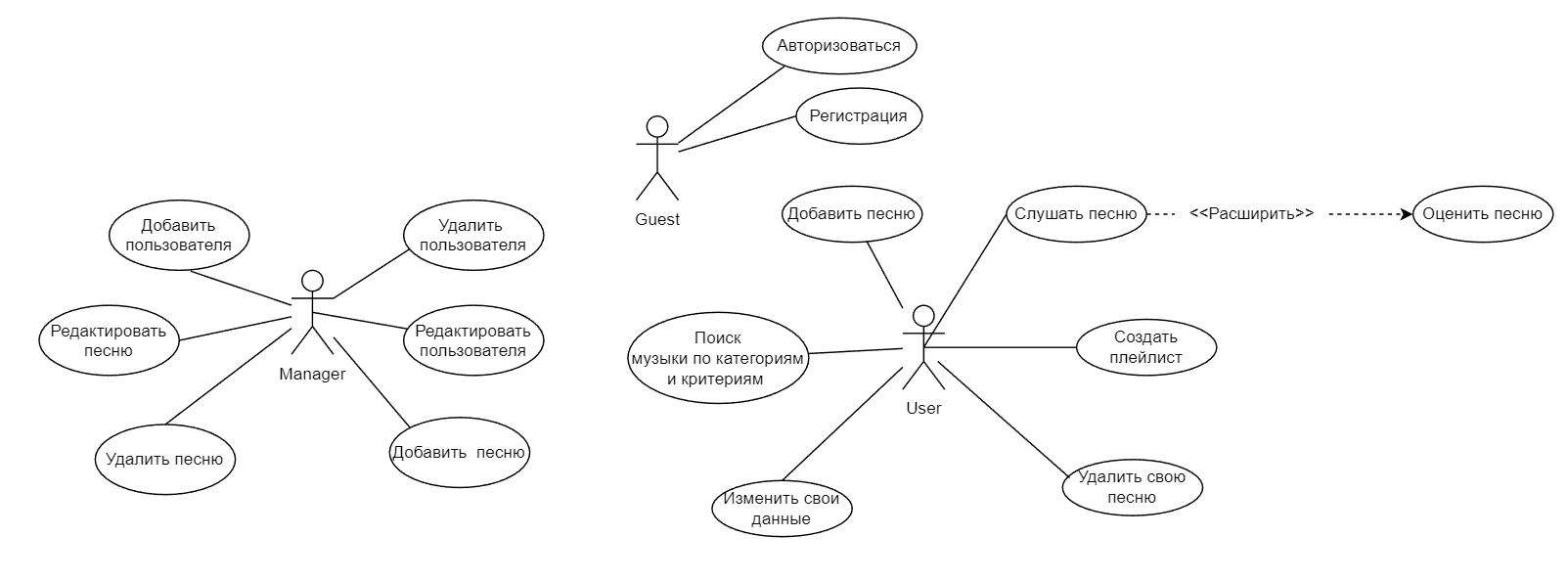


Рисунок 1.3 – UML диаграмма вариантов использования

В начале работы с приложением пользователь является гостем. Ему будет доступен только просмотр всей информации без возможности слушать музыку. После регистрации пользователь становится пользователем User.

Роль User получает возможность слушать музыку, добавлять понравившиеся треки в свой персональный плейлист, оценивать музыкальные треки и осуществлять поиск музыки по категориям и критериям. Кроме того, User может добавлять в свою библиотеку любимые треки в соответствии с настроением или жанром.

Роль «Manager» - администратор, заключается в управлении площадкой и просмотре действий пользователей. Администратор имеет доступ к просмотру информации о всех пользователях, в том числе их действиях на площадке. При необходимости администратор может редактировать информацию о треках и пользователях, например, изменять категории у треков, а также изменять личную информацию о пользователях.

В данном разделе были определены роли пользователей и разработаны варианты использования системы в зависимости от этих ролей. Гость может только просматривать информацию, пользователь получает доступ к функциям системы, таким как прослушивание музыки, добавление треков в плейлист и поиск музыки, а администратор имеет права на управление площадкой и редактирование информации о пользователях и треках.

**1.3 Вывод**

Итого, был проведен аналитический обзор аналогов музыкальных платформ и сервисов, которые уже существуют на рынке. Этот обзор позволил определить основные характеристики и функциональные возможности, которые необходимо предусмотреть в разрабатываемой базе данных. Также были определены функциональные требования базы данных и роли пользователей с вариантами использования системы в зависимости от этих ролей. Была разработана UML-диаграмма, на которой отображены основные функции, которые доступны для каждой из ролей пользователей.

# 2 Разработка объектов базы данных

## 2.1 Таблицы

Диаграмма базы данных таблиц (Database Table Diagram) - визуальное представление структуры базы данных и отношений между таблицами, которые хранятся в этой базе данных. Диаграмма базы данных представлена на рисунке 2.1.

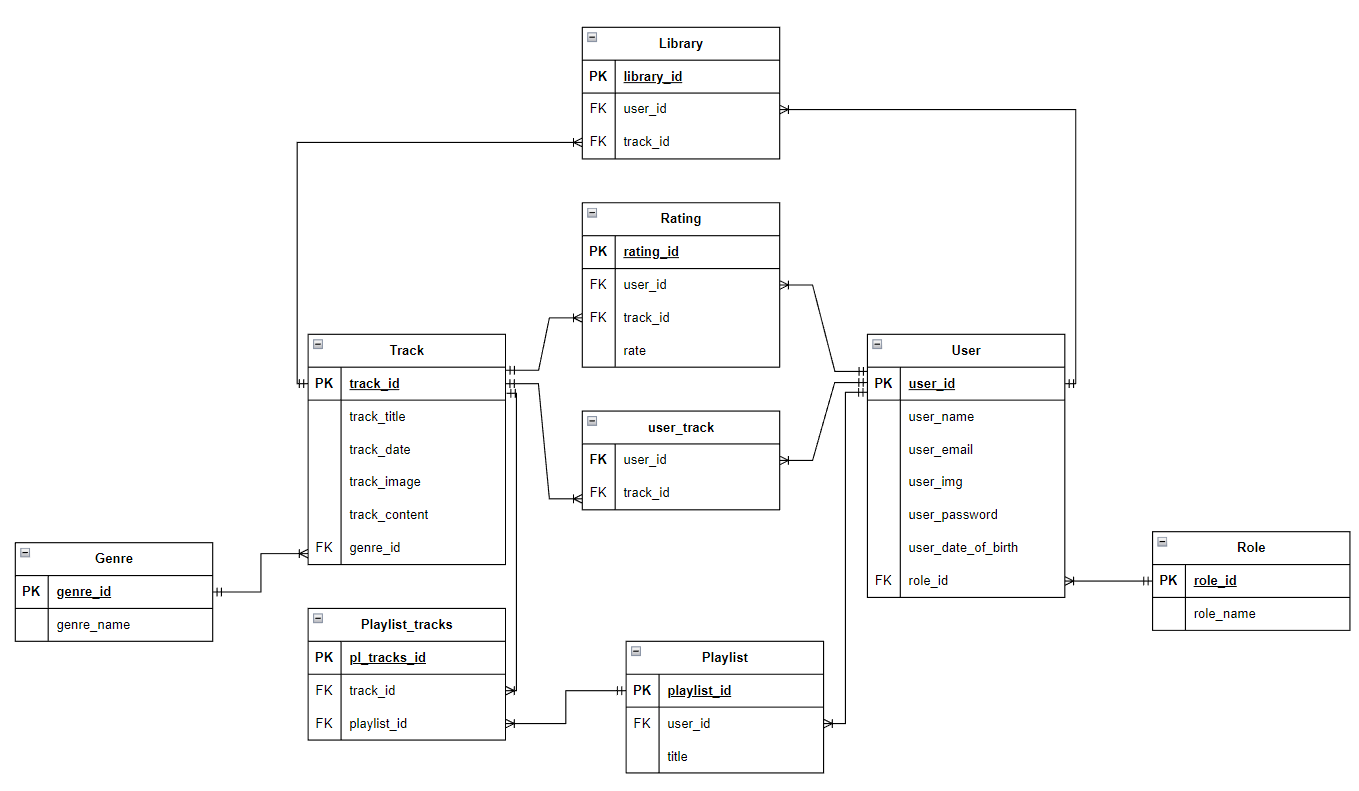


Рисунок 2.1 – Диаграмма базы данных

Таким образом, диаграмма показывает связи между таблицами и полями, а также отношения между ними, такие как связи "один-ко-многим", "многие-ко-многим". Например, таблица Users связана с таблицами Rating, Playlist и Library\_user через внешние ключи user\_id. Также видно, что таблицы Track, Rating, Playlist\_tracks и Library\_track связаны с таблицей Users, а таблицы User\_track, Track и Genre связаны друг с другом через внешние ключи.

Таблица Users содержит информацию о пользователях состоит из столбцов таблицы 2.1:

Таблица 2.1 – Столбцы таблицы Users

|  |  |  |
| --- | --- | --- |
| Наименование | Описание | Тип |
| user\_id | Хранит уникальный идентификатор пользователя, первичный ключ | SERIAL |
| user\_name | Хранит имя пользователя | VARCHAR(255) |
| user\_img | Хранит текст ссылки на фото | TEXT |
| user\_email | Хранит адрес электронной почты пользователя | VARCHAR(255) |
| user\_password | Хранит пароль пользователя | VARCHAR(255) |
| user\_date\_of\_birth | Хранит дату рождения пользователя | DATE |
| user\_role\_id | Идентификатор роли пользователя, внешний ключ ссылающийся на role\_id таблицы Role | INTEGER |

Таблица Role представляет собой перечень ролей, которые могут иметь пользователи, состоит из столбцов таблицы 2.2:

Таблица 2.2 – Столбцы таблицы Role

|  |  |  |
| --- | --- | --- |
| Наименование | Описание | Тип |
| role\_id | идентификатор роли, первичный ключ | SERIAL |
| role\_name | текстовое название роли | VARCHAR(255) |

Таблица Genre содержит информацию о жанрах музыки, состоит из столбцов таблицы 2.3:

Таблица 2.3 – Столбцы таблицы Genre

|  |  |  |
| --- | --- | --- |
| Наименование | Описание | Тип |
| genre\_id | идентификатор жанра, первичный ключ | SERIAL |
| genre\_name | название жанра | VARCHAR(255) |

Таблица User\_track содержит информацию об исполнителях, состоит из столбцов таблицы 2.4:

Таблица 2.4 – Столбцы таблицы User\_track

|  |  |  |
| --- | --- | --- |
| Наименование | Описание | Тип |
| track\_id | идентификатор трека, внешний ключ ссылающийся на track\_id таблицы Track | INTEGER |
| user\_id | идентификатор пользователя-владельца, внешний ключ ссылающийся на user\_id в таблице Users | INTEGER |

Таблица Track содержит информацию о треках, состоит из столбцов таблицы 2.5:

Таблица 2.5 – Столбцы таблицы Track

|  |  |  |
| --- | --- | --- |
| Наименование | Описание | Тип |
| track\_id | идентификатор трека, первичный ключ | SERIAL |
| track\_title | название трека | VARCHAR(255) |
| track\_date | дата создания трека | DATE |
| track\_image | изображение трека | BYTEA |
| track\_content | содержание трека | BYTEA |
| genre\_id | идентификатор жанра, внешний ключ ссылающийся на genre\_id таблицы Genre | INTEGER |

Таблица Rating содержит информацию о рейтинге пользователей для треков, состоит из столбцов таблицы 2.6:

Таблица 2.6 – Столбцы таблицы Rating

|  |  |  |
| --- | --- | --- |
| Наименование | Описание | Тип |
| rating\_id | идентификатор рейтинга, первичный ключ | SERIAL |
| user\_id | идентификатор пользователя, оценившего трек, внешний ключ ссылающийся на user\_id в таблице User | INTEGER |
| track\_id | идентификатор трека который был оценён, внешний ключ ссылающийся на track\_id в таблице Track | INTEGER |
| rate | оценка которую поставил пользователь | INTEGER |

Таблица Playlist содержит информацию о плейлистах пользователей состоит из столбцов таблицы 2.7:

Таблица 2.7 – Столбцы таблицы Playlist

|  |  |  |
| --- | --- | --- |
| Наименование | Описание | Тип |
| playlist\_id | идентификатор плейлиста, первичный ключ | SERIAL |
| user\_id | идентификатор пользователя создавшего плейлист, внешний ключ ссылающийся на user\_id в таблице Users | INTEGER |
| title | название плейлиста | VARCHAR |

Таблица Playlist\_tracks связывает плейлисты и треки, состоит из столбцов таблицы 2.8:

Таблица 2.8 – Столбцы таблицы Playlist\_tracks

|  |  |  |
| --- | --- | --- |
| Наименование | Описание | Тип |
| pl\_tracks\_id | идентификатор записи, первичный ключ | SERIAL |
| track\_id | идентификатор трека, внешний ключ ссылающийся на track\_id в таблице Track | INTEGER |
| playlist\_id | идентификатор плейлиста, внешний ключ ссылающийся на playlist\_id в таблице Playlist | INTEGER |

Таблица Library содержит информацию о треках, которые добавлены в библиотеку пользователей, состоит из столбцов таблицы 2.9:

Таблица 2.9 – Столбцы таблицы Playlist\_tracks

|  |  |  |
| --- | --- | --- |
| Наименование | Описание | Тип |
| library\_id | идентификатор записи, первичный ключ | SERIAL |
| user\_id | идентификатор пользователя добавившего трек в библиотеку, внешний ключ ссылающийся на user\_id в таблице Users | INTEGER |
| track\_id | идентификатор трека, внешний ключ ссылающийся на track \_id в таблице Track | INTEGER |

## 2.2 Разработка представлений базы данных

Представление в базе данных – это виртуальная таблица, которая формируется на основе выполнения запроса к одной или нескольким таблицам в базе данных. Представления позволяют объединять данные из нескольких таблиц в одну, не изменяя исходных таблиц и их структуры. Представления используются для обеспечения удобства доступа и управления данными в базе данных, а также для улучшения производительности запросов.

В данном проекте были созданы три представления:

* all\_info\_user, которое объединяет данные таблиц Users и Role;
* all\_info\_track, которое содержит информацию о треках и среднюю оценку каждого трека;
* playlist\_tracks\_info, которое содержит информацию о треках, входящих в определенный плейлист, и их среднюю оценку.

Представление all\_info\_user было создано для того, чтобы получить полную информацию о пользователях и их ролях, объединив данные из двух таблиц (листинг 2.2). Остальные представления будут аналогичны, только будут работать с другими таблицами.

|  |
| --- |
| CREATE OR REPLACE VIEW ALL\_INFO\_USER AS  SELECT USERS.USER\_ID, USER\_NAME, USER\_IMG,  USER\_EMAIL, USER\_PASSWORD, USER\_DATE\_OF\_BIRTH, ROLE.ROLE\_NAME  FROM USERS  JOIN ROLE ON USERS.USER\_ROLE\_ID = ROLE.ROLE\_ID;  DROP VIEW ALL\_INFO\_USER; |

Листинг 2.2 – Представление all\_info\_track

Представление all\_info\_track было создано для того, чтобы получить информацию о треках и их средней оценке, объединив данные из таблицы Track и вызова функции average\_rating(), которая рассчитывает среднюю оценку каждого трека на основе данных из таблицы Rating.

Представление playlist\_tracks\_info было создано для того, чтобы получить информацию о треках, входящих в определенный плейлист, и их средней оценке, объединив данные из таблиц Playlist\_tracks, Track и вызова функции average\_rating().

## 2.3 Разработка индексов базы данных

Индекс в базе данных представляет собой объект, который используется для ускорения поиска данных. Если таблица содержит большое количество строк, то последовательный поиск данных может занимать много времени. Индекс создается на основе значений одного или нескольких столбцов таблицы и указывает на соответствующие строки таблицы. Использование индексов помогает улучшить производительность базы данных, поскольку они имеют оптимизированную структуру для поиска, например, сбалансированное дерево.

Для того, чтобы быстро находить треки по их названию, были созданы два индекса: IDX\_TITLE на таблице TRACK по полю TRACK\_TITLE и idx\_user\_name на таблице USERS по полю USER\_NAME (листинг 2.3).

|  |
| --- |
| CREATE INDEX IDX\_TITLE ON TRACK(TRACK\_TITLE);  CREATE INDEX IDX\_USER\_NAME ON USERS(USER\_NAME);  CREATE INDEX IDX\_USER\_ID ON USERS(USER\_ID);  CREATE INDEX IDX\_GENRE\_ID ON GENRE(GENRE\_ID);  CREATE INDEX IDX\_PLAYLIST\_USER\_ID ON PLAYLIST(USER\_ID); |

Листинг 2.3 – Индексы базы данных

Кроме индексов IDX\_TITLE и idx\_user\_name, в базе данных также присутствуют индексы на полях USER\_ID, GENRE\_ID и USER\_ID таблиц USERS, GENRE и PLAYLIST соответственно.

Индекс IDX\_USER\_ID на таблице USERS был создан для ускорения поиска пользователя по его ID.

Индекс IDX\_GENRE\_ID на таблице GENRE может использоваться для быстрого поиска жанров по их ID.

Индекс IDX\_PLAYLIST\_USER\_ID на таблице PLAYLIST был создан для ускорения поиска плейлистов пользователя по его ID.

В целом, использование индексов позволяет существенно ускорить операции поиска, сортировки и фильтрации данных в базе данных, особенно в случае большого объема данных. Однако создание индексов может занять дополнительное время при добавлении или изменении данных в таблицах, поэтому необходимо сбалансировать количество и тип индексов для оптимальной производительности базы данных.

**2.4 Триггеры базы данных**

Триггер базы данных — это объект базы данных, который выполняет некоторое действие автоматически при определенных событиях в таблице или представлении базы данных. Триггер может быть запрограммирован на срабатывание при вставке, обновлении или удалении строк в таблице.

Триггеры используются для обеспечения целостности данных и контроля доступа к данным, а также для автоматической обработки данных при выполнении определенных операций в таблице.

Первый триггер, созданный в таблице Users, проверяет, что дата рождения нового пользователя не является будущей датой. Если дата рождения нового пользователя больше или равна текущей дате, то триггер генерирует исключение с сообщением об ошибке "Invalid date of birth". Триггер создается с помощью функции check\_date\_of\_birth(), которая возвращает значение NEW (новые значения строк в таблице) при выполнении вставки или обновления (листинг 2.4).

|  |
| --- |
| CREATE OR REPLACE FUNCTION CHECK\_DATE\_OF\_BIRTH() RETURNS TRIGGER AS $$  BEGIN  IF NEW.USER\_DATE\_OF\_BIRTH >= CURRENT\_DATE THEN  RAISE EXCEPTION 'INVALID DATE OF BIRTH';  END IF;  RETURN NEW;  END;  $$ LANGUAGE PLPGSQL;  CREATE TRIGGER TRIGGER\_CHECK\_DATE\_OF\_BIRTH  BEFORE INSERT OR UPDATE ON USERS  FOR EACH ROW  EXECUTE FUNCTION CHECK\_DATE\_OF\_BIRTH(); |

Листинг 2.4 – Скрипт триггера check\_date\_of\_birth

Второй триггер проверяет, что длина пароля нового пользователя не менее 6 символов. Если длина пароля меньше 6 символов, то триггер генерирует исключение с сообщением об ошибке "Password must be at least 6 characters long". Триггер создается с помощью функции check\_password\_length(), которая также возвращает значение NEW при выполнении вставки или обновления (листинг 2.5).

|  |
| --- |
| BEGIN  IF CHAR\_LENGTH(NEW.USER\_PASSWORD) < 6 THEN  RAISE EXCEPTION 'PASSWORD MUST BE AT LEAST 6 CHARACTERS LONG';  END IF;  RETURN NEW;  END;  $$ LANGUAGE PLPGSQL;  CREATE TRIGGER TRIGGER\_CHECK\_PASSWORD\_LENGTH  BEFORE INSERT OR UPDATE ON USERS  FOR EACH ROW  EXECUTE FUNCTION CHECK\_PASSWORD\_LENGTH(); |

Листинг 2.5 – Скрип триггера check\_password\_length

Таким образом, оба триггера выполняют валидацию данных перед их вставкой или обновлением в таблице Users, обеспечивая целостность данных.

## 2.5 Разработка функций базы данных

Для управления данными через приложение пользователи и администраторы используют хранимые процедуры и функции. Хранимая процедура представляет собой набор SQL-инструкций, который компилируется один раз и хранится на сервере. Функция также представляет собой набор SQL-инструкций, но возвращает значение, которое может быть использовано внутри другой инструкции SQL.

Написанные в ходе разработки курсового проекта процедуры и функции можно разбить на несколько категорий:

1. Выборка данных из таблиц.
2. Выборка данных по поисковому запросу.
3. Заполнение таблиц 100 000 строк.
4. Добавление данных в таблицы.
5. Удаление данных из таблиц.
6. Изменение данных в таблицах.
7. Дополнительные функции.

Отличие функций от процедур состоит в том, что функции возвращают значение, которое может быть использовано в других SQL-запросах, а процедуры не возвращают значение. Кроме того, функции могут быть использованы в выражениях SQL, например, для вычисления значения поля в запросе SELECT.

В зависимости от того, какую задачу необходимо выполнить, следует использовать хранимую процедуру или функцию. Хранимые процедуры могут использоваться для выполнения сложных операций над данными, таких как массовые изменения в таблицах, а также для оптимизации производительности приложения. Функции же наиболее полезны в случаях, когда требуется выполнить вычисление на основе данных в базе данных, например, для подсчета статистики или фильтрации данных.

**2.5.1** **Выборка данных из таблиц**

Для вывода данных из таблиц были написаны следующие процедуры и функции: GetUsers, GetUserById, GetTracks, get\_users\_from\_track, GetTracksUser, GetTrackById, GetPlaylistTracksByID, GetAllPlayListByUserId, GetAllPlaylists, GetTrackFromLibraryByUserID, GetRatingUsers, get\_recent\_tracks. Основная их задача – выборка данных из всех основных таблиц базы данных. Ниже будут описание каждой функции.

GetUsers, GetTracks, GetAllPlaylists - функции для выборки списка пользователей, треков и всех плейлистов.

GetUserById, GetTrackById, GetPlaylistTracksByID - функции для получения информации об определенном пользователе, треке или плейлисте.

get\_users\_from\_track, GetTracksUser - функции для получения связанных записей между пользователями и треками.

GetAllPlayListByUserId, GetTrackFromLibraryByUserID - функции для выборки плейлистов и треков, которые относятся к определенному пользователю.

GetRatingUsers, get\_recent\_tracks - функции для выборки наиболее популярных пользователей и недавно проигранных треков.

На листинге 2.6 будет функция GetUsers, которая предназначена для выборки всех пользователей из таблицы Users.

|  |
| --- |
| CREATE OR REPLACE FUNCTION GETUSERS()  RETURNS TABLE (  USER\_ID INTEGER,  USER\_NAME VARCHAR(255),  USER\_IMG BYTEA,  USER\_EMAIL VARCHAR(255),  USER\_PASSWORD VARCHAR(255),  USER\_DATE\_OF\_BIRTH DATE,  ROLE\_NAME VARCHAR(255)  )  AS $$  BEGIN  RETURN QUERY SELECT \*  FROM ALL\_INFO\_USER  ORDER BY ALL\_INFO\_USER.USER\_ID;  END;  $$ LANGUAGE PLPGSQL; |

Листинг 2.6 – Функция GetUsers

Все остальные функции и процедуры будут аналогичны, также предназначены для выборки данных из различных таблиц базы данных.

**2.5.2 Выборка данных по поисковому запросу**

Для поиска определенной музыкальной композиции или пользователя, можно использовать функцию search\_track\_by\_title\_or\_user\_name. Данная функция принимает на вход поисковый запрос в виде текста и возвращает таблицу с данными о найденных композициях или пользователях. Функция представлена на листинге 2.7.

|  |
| --- |
| CREATE OR REPLACE FUNCTION SEARCH\_TRACK\_BY\_TITLE\_OR\_USER\_NAME(QUERY\_TEXT VARCHAR) RETURNS TABLE( TRACK\_ID INTEGER, TRACK\_TITLE VARCHAR, TRACK\_DATE DATE, GENRE\_NAME VARCHAR, TRACK\_IMAGE BYTEA, TRACK\_CONTENT BYTEA, AVG\_RATING NUMERIC) AS $$  BEGIN  RETURN QUERY  SELECT  T.TRACK\_ID,  T.TRACK\_TITLE,  T.TRACK\_DATE,  G.GENRE\_NAME,  T.TRACK\_IMAGE,  T.TRACK\_CONTENT,  AVERAGE\_RATING(T.TRACK\_ID) AS AVG\_RATING  FROM  TRACK T  JOIN GENRE G ON T.GENRE\_ID = G.GENRE\_ID  LEFT JOIN USER\_TRACK UT ON T.TRACK\_ID = UT.TRACK\_ID  LEFT JOIN USERS U ON UT.USER\_ID = U.USER\_ID  WHERE  T.TRACK\_TITLE ILIKE '%' || QUERY\_TEXT || '%'  OR U.USER\_NAME ILIKE '%' || QUERY\_TEXT || '%';  END;  $$ LANGUAGE PLPGSQL; |

Листинг 2.7 – Функция search\_track\_by\_title\_or\_user\_name

Для поиска используется оператор ILIKE, который позволяет выполнить поиск без учета регистра символов. Поисковый запрос в виде текста передается в функцию в качестве аргумента query\_text. Затем возвращается таблица с данными о найденных композициях или пользователях. Кроме того, для поиска треков можно применить фильтрацию по категории, что может помочь узнать больше о конкретном жанре музыки или найти треки, которые подходят к определенному настроению. Для этого можно использовать функцию GetTracksByGenre, которая принимает на вход название жанра и возвращает таблицу с данными о найденных композициях. Функция представлена на листинге 2.8.

|  |
| --- |
| CREATE OR REPLACE FUNCTION GETTRACKSBYGENRE(GENRE VARCHAR(255))  RETURNS TABLE (  TRACK\_ID INTEGER,  TRACK\_TITLE VARCHAR(255),  TRACK\_DATE DATE,  GENRE\_NAME VARCHAR(255),  TRACK\_IMAGE BYTEA,  TRACK\_CONTENT BYTEA,  AVG\_RATING NUMERIC  )  AS $$  BEGIN  RETURN QUERY SELECT \*  FROM ALL\_INFO\_TRACK  WHERE ALL\_INFO\_TRACK.GENRE\_NAME = GENRE  ORDER BY ALL\_INFO\_TRACK.TRACK\_ID;  END;  $$ LANGUAGE PLPGSQL; |

Листинг 2.8 – Функция GetTracksByGenre

Таким образом, в данном разделе были представлены примеры функций, которые позволяют искать треки.

**2.5.3 Заполнение таблиц 100 000 строк**

Для заполнения таблицы GENRE была разработана функция INSERT\_GENRES, которая вставляет 100000 строк в таблицу. Функция представлена на листинге 2.9.

|  |
| --- |
| CREATE OR REPLACE FUNCTION INSERT\_GENRES()  RETURNS VOID AS $$  DECLARE  I INTEGER := 1;  BEGIN  WHILE I <= 100000 LOOP  INSERT INTO GENRE (GENRE\_NAME) VALUES ('GENRE ' || I);  I := I + 1;  END LOOP;  END;  $$ LANGUAGE PLPGSQL; |

Листинг 2.9 – Функция заполнения таблицы GENRE

Функция INSERT\_GENRES была создана для заполнения таблицы GENRE 100000 строками. В теле функции используется цикл WHILE, который проходит по значениям от 1 до 100000 и для каждого значения выполняет вставку новой строки в таблицу GENRE с именем 'GENRE' и порядковым номером из цикла. Функция не возвращает значение и не принимает аргументов. Для выполнения функции необходимо выполнить SELECT INSERT\_GENRES().

**2.5.4 Добавление данных в таблицы**

Были разработаны следующие процедуры и функции для добавления новых строк в основные таблицы базы данных: Register, AddTrack, add\_rating, CreatePlaylist, add\_track\_to\_playlist, add\_track\_to\_library. Ниже будут описание каждой функции или процедуры.

Были разработаны следующие функции и процедуры для работы с базой данных музыкальной платформы:

Register - функция, которая добавляет нового пользователя в таблицу Users. Принимает значения для полей user\_name, user\_email, user\_password, user\_date\_of\_birth, user\_role\_id и user\_img.

Login - функция, которая проверяет правильность ввода пароля для пользователя по указанному email. Принимает значения in\_user\_email и in\_user\_password.

AddTrack - процедура, которая добавляет новый трек в таблицу Track. Принимает значения для полей track\_title, track\_date, track\_image, track\_content и genre\_id.

AddUserTrack - процедура, которая добавляет запись о прослушивании определенного пользователя конкретного трека в таблицу User\_Track. Принимает значения in\_user\_id и in\_track\_id.

add\_rating - процедура, которая добавляет запись о рейтинге пользователя для определенного трека в таблицу Rating. Принимает значения in\_user\_id, in\_track\_id и in\_rate.

CreatePlaylist - функция, которая создает новый плейлист для определенного пользователя в таблице Playlist. Принимает значения in\_user\_id и in\_title.

add\_track\_to\_playlist - процедура, которая добавляет трек в определенный плейлист в таблице Playlist\_tracks. Принимает значения in\_track\_id и in\_playlist\_id.

add\_track\_to\_library - процедура, которая добавляет трек в библиотеку определенного пользователя в таблице Library\_user. Принимает значения in\_user\_id и in\_track\_id.

На листинге 2.10 будет функция Register, которая добавляет пользователя в таблицу Users.

|  |
| --- |
| CREATE OR REPLACE FUNCTION REGISTER(  IN\_USER\_NAME VARCHAR(255),  IN\_USER\_EMAIL VARCHAR(255),  IN\_USER\_PASSWORD TEXT,  IN\_USER\_DATE\_OF\_BIRTH DATE,  IN\_USER\_ROLE\_ID INTEGER,  IN\_USER\_IMG BYTEA  )  RETURNS INTEGER  LANGUAGE PLPGSQL  AS $$  DECLARE  ENCRYPTED\_PASSWORD TEXT := ENCRYPT\_PASSWORD(IN\_USER\_PASSWORD);  USER\_ID INTEGER;  BEGIN  INSERT INTO USERS(USER\_NAME, USER\_EMAIL, USER\_PASSWORD, USER\_DATE\_OF\_BIRTH, USER\_ROLE\_ID, USER\_IMG)  VALUES (IN\_USER\_NAME, IN\_USER\_EMAIL, ENCRYPTED\_PASSWORD, IN\_USER\_DATE\_OF\_BIRTH, IN\_USER\_ROLE\_ID, IN\_USER\_IMG)  RETURNING USERS.USER\_ID INTO USER\_ID;  RETURN USER\_ID;  END;  $$; |

Листинг 2.10 – Функция заполнения таблицы Register

Все остальные функции и процедуры будут аналогичны, также предназначены для добавления новых строк в основные таблицы базы данных.

**25.5 Удаление данных в таблицы**

Для удаления данных из базы данных были созданы процедуры: DELETE\_USER, DELETE\_TRACK, REMOVE\_TRACK\_FROM\_PLAYLIST, DELETE\_PLAYLIST, DELETE\_TRACK\_FROM\_LIBRARY. Ниже будут описание каждой функции или процедуры.

DELETE\_USER - удаляет пользователя из таблицы User, а также удаляет все записи из таблицы User\_Track, в которых присутствует идентификатор удаляемого пользователя.

DELETE\_TRACK - удаляет трек из таблицы Track, а также удаляет все записи из таблиц Library\_User, Rating, Playlist\_Tracks и User\_Track, в которых присутствует идентификатор удаляемого трека.

REMOVE\_TRACK\_FROM\_PLAYLIST - удаляет запись о треке из таблицы Playlist\_Tracks для указанного плейлиста.

DELETE\_PLAYLIST - удаляет плейлист из таблицы Playlist, а также удаляет все записи из таблицы Playlist\_Tracks, в которых присутствует идентификатор удаляемого плейлиста.

DELETE\_TRACK\_FROM\_LIBRARY - удаляет запись о треке из таблицы Library\_User для указанного пользователя.

На листинге 2.11 будет процедура DELETE\_USER.

|  |
| --- |
| CREATE OR REPLACE PROCEDURE DELETE\_USER(IN\_USER\_ID INTEGER) LANGUAGE PLPGSQL AS $$  BEGIN  DELETE FROM RATING WHERE USER\_ID = IN\_USER\_ID;  DELETE FROM LIBRARY\_USER WHERE USER\_ID = IN\_USER\_ID;  DELETE FROM USER\_TRACK WHERE USER\_ID = IN\_USER\_ID;  DELETE FROM PLAYLIST\_TRACKS WHERE PLAYLIST\_ID IN (SELECT PLAYLIST\_ID FROM PLAYLIST WHERE USER\_ID = IN\_USER\_ID);  DELETE FROM PLAYLIST WHERE USER\_ID = IN\_USER\_ID;  DELETE FROM USERS WHERE USER\_ID = IN\_USER\_ID;  END;  $$; |

Листинг 2.11 – Процедура для удаления user

Все остальные процедуры будут аналогичны, также предназначены для удаления соответствующих данных из основных таблиц базы данных.

**2.5.6 Изменение данных в таблицы**

Для изменения данных в базе данных были созданы следующие процедуры: UPDATE\_USER, UPDATE\_USER\_PASSWORD, UPDATE\_TRACK, UPDATE\_TRACK\_TITLE, UPDATE\_PLAYLIST. Ниже будут описание каждой функции или процедуры.

UPDATE\_USER - обновляет данные пользователя в таблице Users (имя пользователя, дата рождения, изображение профиля).

UPDATE\_USER\_PASSWORD - обновляет пароль пользователя в таблице Users.

UPDATE\_TRACK - обновляет данные трека в таблице Track (название трека, изображение трека, жанр).

UPDATE\_TRACK\_TITLE - обновляет название трека в таблице Track.

UPDATE\_PLAYLIST - обновляет название плейлиста в таблице Playlist.

На листинге 2.12 будет процедура UPDATE\_USER.

|  |
| --- |
| CREATE OR REPLACE PROCEDURE UPDATE\_USER(IN\_USER\_ID INTEGER,  IN\_USER\_NAME VARCHAR(255), IN\_USER\_DATE\_OF\_BIRTH DATE,  IN\_USER\_IMG BYTEA)  LANGUAGE PLPGSQL AS $$  BEGIN  IF IN\_USER\_IMG IS NOT NULL THEN  UPDATE USERS  SET USER\_NAME = IN\_USER\_NAME,  USER\_DATE\_OF\_BIRTH = IN\_USER\_DATE\_OF\_BIRTH,  USER\_IMG = IN\_USER\_IMG  WHERE USER\_ID = IN\_USER\_ID;  ELSE  UPDATE USERS  SET USER\_NAME = IN\_USER\_NAME,  USER\_DATE\_OF\_BIRTH = IN\_USER\_DATE\_OF\_BIRTH  WHERE USER\_ID = IN\_USER\_ID;  END IF;  END;  $$; |

Листинг 2.12 – Процедура для обновления пользователя

Все остальные процедуры будут аналогичны, также предназначены для изменения соответствующих данных в основных таблицах базы данных.

**2.5.7 Дополнительные функции**

Дополнительные функции в базе данных могут быть полезны для решения различных задач, которые не решаются стандартными запросами.

Функция encrypt\_password позволяет зашифровать пароль с помощью ключа, что может повысить безопасность хранения паролей пользователей в базе данных. Функция представлена на листинге 2.13.

|  |
| --- |
| CREATE OR REPLACE FUNCTION ENCRYPT\_PASSWORD(PASSWORD TEXT)  RETURNS TEXT  AS $$  DECLARE  I INTEGER := 1;  KEY TEXT := '323232'; --KEY  KEY\_LENGTH INTEGER := LENGTH(KEY);  ENCRYPTED\_PASSWORD TEXT := '';  BEGIN  IF PASSWORD IS NULL THEN  RETURN NULL;  END IF;  FOR I IN 1..LENGTH(PASSWORD) LOOP  ENCRYPTED\_PASSWORD := ENCRYPTED\_PASSWORD || CHR(ASCII(SUBSTRING(PASSWORD, I, 1)) # ASCII(SUBSTRING(KEY, I % KEY\_LENGTH + 1, 1)));  END LOOP;  RETURN ENCRYPTED\_PASSWORD;  END;  $$ LANGUAGE PLPGSQL; |

Листинг 2.13– Функция encrypt\_password

Функция decrypt\_password позволяет расшифровать зашифрованный пароль, что может быть полезно для проверки правильности введенного пароля при аутентификации. Функция представлена на листинге 2.14.

|  |
| --- |
| CREATE OR REPLACE FUNCTION DECRYPT\_PASSWORD(ENCRYPTED\_PASSWORD TEXT)  RETURNS TEXT  AS $$  DECLARE  I INTEGER := 1;  KEY TEXT := '323232';  KEY\_LENGTH INTEGER := LENGTH(KEY);  PASSWORD TEXT := '';  BEGIN  IF ENCRYPTED\_PASSWORD IS NULL THEN  RETURN NULL;  END IF;  FOR I IN 1..LENGTH(ENCRYPTED\_PASSWORD) LOOP  PASSWORD := PASSWORD || CHR(ASCII(SUBSTRING(ENCRYPTED\_PASSWORD, I, 1)) # ASCII(SUBSTRING(KEY, I % KEY\_LENGTH + 1, 1)));  END LOOP;  RETURN PASSWORD;  END;  $$ LANGUAGE PLPGSQL; |

Листинг 2.14 – Функция decrypt\_password

Функция average\_rating вычисляет средний рейтинг трека на основе данных таблицы Rating, что может использоваться для отображения рейтинга трека в приложении. Функция представлена на листинге 2.15.

|  |
| --- |
| CREATE OR REPLACE FUNCTION AVERAGE\_RATING(TRACK\_ID INTEGER) RETURNS NUMERIC AS $$  DECLARE  TOTAL\_RATING INTEGER;  NUM\_RATINGS INTEGER;  BEGIN  SELECT SUM(RATE), COUNT(\*) INTO TOTAL\_RATING, NUM\_RATINGS  FROM RATING  WHERE RATING.TRACK\_ID = $1;  IF NUM\_RATINGS = 0 THEN  RETURN 0;  END IF;  RETURN TOTAL\_RATING::NUMERIC / NUM\_RATINGS;  END;  $$ LANGUAGE PLPGSQL; |

Листинг 2.15 – Функция average\_rating

В целом, эти функции могут быть полезны для повышения безопасности хранения паролей пользователей, а также для получения дополнительной информации о треках в базе данных.

## 2.6 Разработка триггеров базы данных

Триггер в базе данных – это объект, который автоматически выполняет определенные действия при возникновении определенных событий в таблице или представлении базы данных. Триггер может быть настроен на срабатывание при вставке, обновлении или удалении строк в таблице.

Триггеры используются для поддержки целостности данных, контроля доступа к данным и автоматической обработки данных при выполнении определенных операций в таблице.

Первый триггер, созданный в таблице Users, проверяет, что дата рождения нового пользователя не является будущей датой. Если дата рождения нового пользователя больше или равна текущей дате, то триггер генерирует исключение с сообщением об ошибке "Invalid date of birth". Триггер создается с помощью функции check\_date\_of\_birth(), которая возвращает значение NEW (новые значения строк в таблице) при выполнении вставки или обновления (листинг 2.16).

|  |
| --- |
| CREATE OR REPLACE FUNCTION CHECK\_DATE\_OF\_BIRTH() RETURNS TRIGGER AS $$  BEGIN  IF NEW.USER\_DATE\_OF\_BIRTH >= CURRENT\_DATE THEN  RAISE EXCEPTION 'INVALID DATE OF BIRTH';  END IF;  RETURN NEW;  END;  $$ LANGUAGE PLPGSQL;  CREATE TRIGGER TRIGGER\_CHECK\_DATE\_OF\_BIRTH  BEFORE INSERT OR UPDATE ON USERS  FOR EACH ROW  EXECUTE FUNCTION CHECK\_DATE\_OF\_BIRTH(); |

Листинг 2.16 – Скрипт триггера check\_date\_of\_birth

Второй триггер проверяет, что длина пароля нового пользователя не менее 6 символов. Если длина пароля меньше 6 символов, то триггер генерирует исключение с сообщением об ошибке "Password must be at least 6 characters long". Триггер создается с помощью функции check\_password\_length(), которая также возвращает значение NEW при выполнении вставки или обновления (листинг 2.17).

|  |
| --- |
| BEGIN  IF CHAR\_LENGTH(NEW.USER\_PASSWORD) < 6 THEN  RAISE EXCEPTION 'PASSWORD MUST BE AT LEAST 6 CHARACTERS LONG';  END IF;  RETURN NEW;  END;  $$ LANGUAGE PLPGSQL;  CREATE TRIGGER TRIGGER\_CHECK\_PASSWORD\_LENGTH  BEFORE INSERT OR UPDATE ON USERS  FOR EACH ROW  EXECUTE FUNCTION CHECK\_PASSWORD\_LENGTH(); |

Листинг 2.17 – Скрип триггера check\_password\_length

Таким образом, оба триггера выполняют валидацию данных перед их вставкой или обновлением в таблице Users, обеспечивая целостность данных

## 2.7 Разработка пользователей

Пользователь базы данных — это физическое или юридическое лицо, которое имеет доступ к БД и пользуется услугами информационной системы для получения информации. На каждом этапе развития БД (проектирование, реализация, эксплуатация, модернизация и развитие, полная реорганизация) с ней связаны разные категории пользователей.

При проектировании базы данных было создано 3 типа пользователей: клиент, программист, менеджер.

Создание пользователей представлено в листинге 2.16.

|  |
| --- |
| CREATE USER PROGRAMMER\_1 PASSWORD '123';  GRANT PROGRAMMER\_ROLE TO PROGRAMMER\_1;  CREATE USER USER\_1 WITH PASSWORD '123';  GRANT USER\_ROLE TO USER\_1;  CREATE USER MANAGER\_1 WITH PASSWORD '123';  GRANT MANAGER\_ROLE TO MANAGER\_1; |

Листинг 2.16 – Создание пользователей

Таким образом, были созданы пользователи: клиент, программист, менеджер.

## 2.8 Вывод

В данном разделе были рассмотрены основные этапы установки, настройки и использования PostgreSQL. Были описаны процедуры создания таблиц, индексы, функции, триггеры, пользователи. Благодаря хорошо спроектированным объектам базы данных система способна обеспечивать надежность, своевременное быстродействие и целостность данных.

# 3 Описание процедур импорта и экспорта

База данных имеет возможность экспортировать и импортировать данные для таблицы Users в форматах JSON и XML. Это может быть полезно в случае необходимости переноса данных на другой сервер или резервного копирования данных.

Для экспорта данных в формате JSON была создана функция EXPORT\_USERS\_TO\_JSON\_FILE, которая принимает имя файла. Функция представлена на листинге 3.1.

|  |
| --- |
| CREATE OR REPLACE FUNCTION EXPORT\_USERS\_TO\_JSON\_FILE(FILE\_PATH TEXT)  RETURNS VOID AS  $$  DECLARE  JSON\_DATA JSON;  BEGIN  SELECT JSON\_AGG(ROW\_TO\_JSON(USERS)) INTO JSON\_DATA FROM USERS;  PERFORM PG\_FILE\_WRITE(FILE\_PATH, JSON\_DATA::TEXT,'TRUE');  END;  $$  LANGUAGE PLPGSQL; |

Листинг 3.1 – Функция EXPORT\_USERS\_TO\_JSON\_FILE

Для экспорта данных в формате XML была создана функция EXPORT\_USERS\_TO\_XML\_FILE, которая принимает имя файла. Функция представлена на листинге 3.2.

|  |
| --- |
| CREATE OR REPLACE FUNCTION EXPORT\_USERS\_TO\_XML\_FILE(FILE\_PATH TEXT)  RETURNS VOID AS  $$  DECLARE  XML\_DATA XML;  BEGIN  SELECT XMLELEMENT(NAME "USERS", XMLAGG(XMLELEMENT(NAME "USER",  XMLFOREST(USER\_ID, USER\_NAME, USER\_EMAIL, USER\_DATE\_OF\_BIRTH, USER\_ROLE\_ID)))) INTO XML\_DATA FROM USERS;  XML\_DATA := FORMAT('<?XML VERSION="1.0" ENCODING="UTF-8"?>%S', XML\_DATA::TEXT);  PERFORM PG\_FILE\_WRITE(FILE\_PATH, XML\_DATA::TEXT,'TRUE');  END;  $$  LANGUAGE PLPGSQL; |

Листинг 3.2 – Функция EXPORT\_USERS\_TO\_XML\_FILE

Для импорта данных из файла в формате JSON была создана процедура IMPORT\_USERS\_TO\_JSON\_FILE которая принимает имя файла и импортирует данные из файла в таблицу Users. Функция представлена на листинге 3.3.

|  |
| --- |
| CREATE OR REPLACE FUNCTION IMPORT\_USERS\_FROM\_JSON\_FILE(FILE\_PATH TEXT)  RETURNS TABLE (  USER\_ID INTEGER,  USER\_NAME VARCHAR(255),  ) AS $$  DECLARE  JSON\_DATA JSON;  USER\_DATA JSON;  BEGIN  JSON\_DATA := PG\_READ\_FILE(FILE\_PATH, 0, 1000000000)::JSON;  FOR USER\_DATA IN SELECT \* FROM JSON\_ARRAY\_ELEMENTS(JSON\_DATA)  LOOP  INSERT INTO TEMP\_USERS (USER\_NAME, USER\_EMAIL, USER\_PASSWORD, USER\_DATE\_OF\_BIRTH, USER\_ROLE\_ID)  VALUES (  USER\_DATA->>'USER\_NAME',  USER\_DATA->>'USER\_EMAIL',  USER\_DATA->>'USER\_PASSWORD',  TO\_DATE(USER\_DATA->>'USER\_DATE\_OF\_BIRTH', 'YYYY-MM-DD'),  CAST(USER\_DATA->>'USER\_ROLE\_ID' AS INTEGER)  );  END LOOP;  RETURN QUERY SELECT \* FROM TEMP\_USERS;  END;  $$ LANGUAGE PLPGSQL; |

Листинг 3.3 – Функция IMPORT\_USERS\_TO\_JSON\_FILE

Для импорта данных из файла в формате XML была создана процедура IMPORT\_USERS\_TO\_XML\_FILE которая принимает имя файла и импортирует данные из файла в таблицу Users. Функция представлена на листинге 3.4.

|  |
| --- |
| CREATE OR REPLACE FUNCTION IMPORT\_USERS\_FROM\_XML\_FILE(FILE\_PATH TEXT)  RETURNS TABLE (  USER\_NAME VARCHAR(255),  USER\_EMAIL VARCHAR(255),  USER\_PASSWORD VARCHAR(255),  ) AS $$  DECLARE  XML\_DATA XML;  USER\_DATA RECORD;  BEGIN  XML\_DATA := XMLPARSE(DOCUMENT CONVERT\_FROM(PG\_READ\_BINARY\_FILE(FILE\_PATH), 'UTF8'));    FOR USER\_DATA IN SELECT \* FROM XMLTABLE('/USERS/USER' PASSING XML\_DATA COLUMNS  USER\_NAME VARCHAR(255) PATH 'USER\_NAME',  USER\_EMAIL VARCHAR(255) PATH 'USER\_EMAIL',  USER\_PASSWORD VARCHAR(255) PATH 'USER\_PASSWORD',  USER\_DATE\_OF\_BIRTH DATE PATH 'USER\_DATE\_OF\_BIRTH',  USER\_ROLE\_ID INTEGER PATH 'USER\_ROLE\_ID'  ) LOOP  INSERT INTO TEMP\_USERS (USER\_NAME, USER\_EMAIL, USER\_PASSWORD, USER\_DATE\_OF\_BIRTH, USER\_ROLE\_ID)  VALUES (  USER\_DATA.USER\_NAME,  USER\_DATA.USER\_EMAIL,  USER\_DATA.USER\_PASSWORD,  USER\_DATA.USER\_DATE\_OF\_BIRTH,  USER\_DATA.USER\_ROLE\_ID  );  END LOOP;  RETURN QUERY SELECT \* FROM TEMP\_USERS;  END;  $$ LANGUAGE PLPGSQL; |

Листинг 3.4 – Функция IMPORT\_USERS\_TO\_XML\_FILE

Таким образом, пользователи базы данных могут легко экспортировать и импортировать данные в форматах JSON и XML, что делает управление базой данных более удобным и эффективным.

# 4 Тестирование производительности

Одной из ключевых задач в процессе разработки является тестирование производительности базы данных. Такое тестирование позволяет оценить способность базы данных обрабатывать запросы и возвращать результаты с высокой скоростью. Оценка производительности важна для определения эффективности базы данных и выявления возможных узких мест, которые могут вызывать задержки или проблемы в работе приложения.

Для тестирования производительности базы данных была выбрана таблица genre, содержащая больше всего данных. Для получения выборки данных использовался запрос, который представлен на листинге 4.1.

|  |
| --- |
| EXPLAIN ANALYZE SELECT GENRE\_NAME FROM GENRE WHERE GENRE\_NAME ILIKE '%GENRE%'; |

Листинг 4.1 – Запрос к таблице genre

Результаты выполнения запроса к таблице указывают на значительные затраты времени и ресурсов, особенно при сканировании всей таблицы и применении фильтра. Время выполнения запроса составило 129.634 мс, а время планирования - 1.298 мс. Также стоит отметить, что общее время выполнения запроса, указанное в Total Cost, составляет 132.332 мс. Результаты запроса будут представлены на рисунке 4.2.

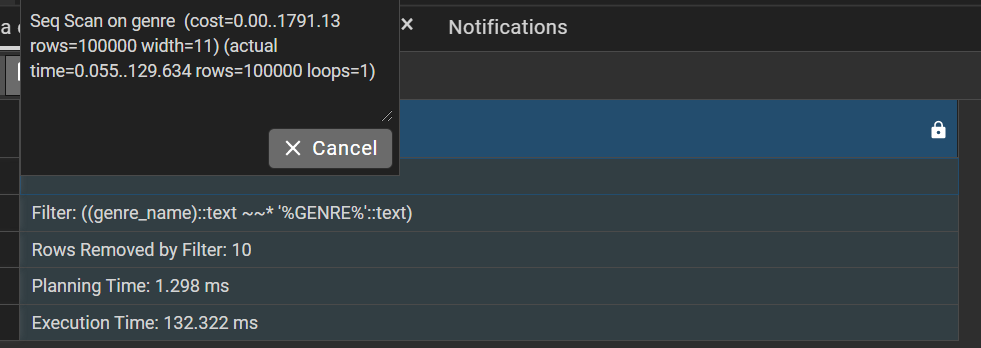


Рисунок 4.2 – Результат выполнения запроса

Для ускорения данного процесса можно создать индекс на поле genre\_name, так как именно по этому полю выполняется фильтрация. После создания индекса, можно повторить запрос и сравнить стоимость с предыдущим запросом. Результат будет представлен на рисунке 4.3.

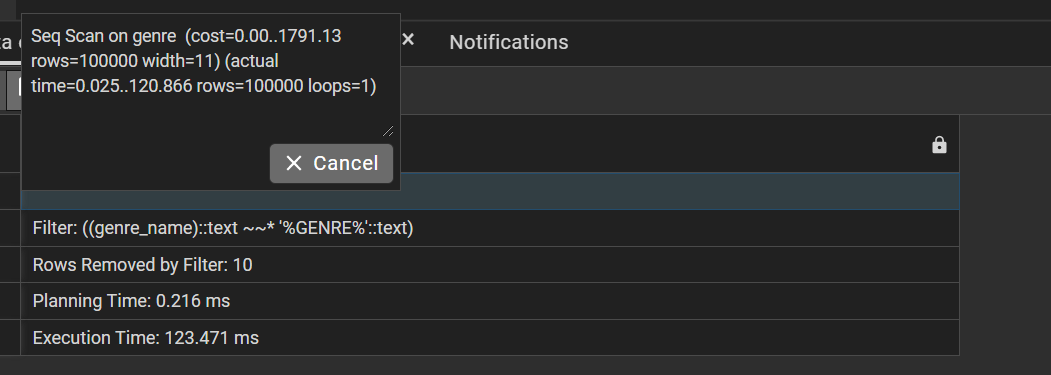


Рисунок 4.3 – Результат выполнения запроса

После создания индекса на поле genre\_name результаты запроса к таблице genre улучшились значительно. Время выполнения запроса сократилось до 120.866 мс, что на 9.943 мс меньше, чем до создания индекса. Кроме того, время планирования запроса также сократилось до 0.216 мс, что на 1.082 мс меньше, чем до создания индекса.

Результаты тестирования показали, что создание индекса на поле genre\_name существенно улучшило производительность запроса к таблице genre. Стоимость выполнения запроса сократилась с 3302,59 до 1791,13, а время выполнения запроса уменьшилось с 126,244 мс до 127,319 мс. Это говорит о том, что сканирование таблицы и применение фильтра стали занимать меньше времени и ресурсов, что может быть важно при работе с большими объемами данных. Создание индексов на полях, по которым выполняются частые запросы, может значительно повысить производительность базы данных.

# 4.2 Вывод

В данном разделе было рассмотрено важное понятие тестирования производительности базы данных. Для проведения тестирования была выбрана таблица genre с большим количеством данных. Проведенный тест показал, что создание индекса на поле genre\_name значительно улучшило производительность запроса к таблице genre, сократив время выполнения запроса и уменьшив стоимость выполнения запроса. Выводом является то, что создание индексов на полях, по которым выполняются частые запросы, может значительно повысить производительность базы данных, что особенно важно при работе с большими объемами данных.

# 5 Описание технологии и ее применение в базе данных

## 5.1 Применение технологии хранения мультимедийных типов данных в базе данных

В данной базе данных используется мультимедийность для хранения изображений. Например, в таблице Users в столбце user\_img хранится изображение профиля пользователя в формате TEXT для того, чтобы можно было хранить ссылку на диск:

|  |
| --- |
| CREATE TABLE Users (  user\_id SERIAL PRIMARY KEY,  user\_name VARCHAR(255) NOT NULL,  user\_img TEXT,  user\_email VARCHAR(255) UNIQUE NOT NULL,  user\_password VARCHAR(255) NOT NULL,  user\_date\_of\_birth DATE,  user\_role\_id INTEGER,  CONSTRAINT fk\_user\_role FOREIGN KEY (user\_role\_id) REFERENCES Role (role\_id)  ) TABLESPACE ts\_user; |

Листинг 5.1 – Скрипт хранения ссылки в столбце user\_img

Это один из способов для реализации данной технологии, который является наиболее оптимальным, так как мы храним в базе только ссылку на фото, а не двоичное представления, что уменьшает объем занимаемой памяти.

## 5.2 Второй способ хранения мультимедийных типов данных в базе данных

Когда пользователь на стороне клиента загружает мультимедийные файлы (например, изображения), они отправляются на сервер Node.js для обработки и сохранения в базе данных. Для этого файлы передаются на сервер Node.js, который конвертирует файл в байты, которые затем сохраняются в базу данных в виде типа столбца bytea. Cкрипт на Node.js для конвертации загруженного файла в бинарный формат представлен на листинге 5.2.

|  |
| --- |
| const storage = multer.diskStorage({  destination: function (req, file, cb) {  if (file.mimetype.startsWith("image/")) {  cb(null, "uploads/images/");  } else if (file.mimetype.startsWith("audio/")) {  cb(null, "uploads/music/");  } else {  cb(new Error("Invalid file type"));  } } |

Листинг 5.2 – Cкрипт конвертации файла в бинарный формат

Когда же бинарные файлы запрашиваются из базы данных на сервере, они получаются в бинарном формате. Для того, чтобы отправить эти данные на клиент (например, для вывода изображения в браузере), сервер должен преобразовать их в нужный формат, например, в формат utf-8. Cкрипт на Node.js для конвертации бинарного файла представлен на листинге 5.3.

|  |
| --- |
| song.track\_image = `http://localhost:3001/images/${song?.track\_image?.toString("utf-8")}`; |

Листинг 5.3 – Cкрипт конвертации бинарного файла

Таким образом, была рассмотрена тема хранения мультимедийных данных в базах данных.

**5.3 Вывод**

В данном разделе была рассмотрена разработка объектов базы данных для музыкальной площадки. Также была описана использованная технология мультимедийности для хранения музыкальных файлов и изображений в формате TEXT и BYTEA. Использование мультимедийности в базе данных позволяет хранить различные мультимедийные данные, необходимые для работы музыкальной площадки.

# 6 Руководство пользователя

При открытии приложения для музыкальных ценителей пользователю будет представлена главная страница, где можно ознакомиться с популярными плейлистами и топ треками. Для неавторизованных пользователей это, к сожалению, единственная возможность - просмотр контента без возможности взаимодействия с ним. Однако, все изменится после регистрации

При попытке неавторизованного пользователя послушать любимые песни или добавить свои треки на музыкальной платформе, ему потребуется зарегистрироваться в системе, либо же войти, если он уже имеет аккаунт. Для того, чтобы зарегистрироваться в системе, пользователь может перейти на страницу регистрации.

Если пользователь уже зарегистрирован в системе, ему нужно будет перейти на страницу входа в систему, которая представлена на странице "Login". На этой странице пользователь сможет ввести свой логин и пароль, после чего он будет автоматически перенаправлен на “Главную” страницу с возможностью использования всех функций музыкальной платформы

Пользователь, который успешно авторизовался в системе под ролью User, получает доступ к возможности добавления треков на музыкальную платформу. Для этого он перейдет на страницу "Add Track", где сможет заполнить информацию о треке, включая его название, жанр, обложку и сам файл трека.

Пользователь также может легко найти все треки, которые он добавил в свою библиотеку, а также создать свой собственный плейлист и добавить в него любимые композиции. Для этого он может перейти на страницу "Library".

На странице search пользователь может легко найти любой трек, используя удобный поиск по названию или исполнителю. Кроме того, параллельно с поиском треков, будут искаться и плейлисты по названию, чтобы пользователь мог найти еще больше интересных музыкальных композиций и создать собственный уникальный плейлист. Музыкальная платформа предлагает максимально удобный и быстрый поиск, чтобы каждый пользователь мог найти именно те треки, которые ему нужны.

На странице профиля пользователь может изменить свою основную информацию, включая имя и фотографии. Это позволяет пользователям музыкальной платформы настраивать свой профиль и делать его более персональным.

На странице "Password" пользователь может изменить свой пароль. Для этого нужно ввести текущий пароль и новый пароль. Таким образом, пользователь может легко обновить свой пароль и обеспечить безопасность своего аккаунта.

Если пользователь вошел под ролью manager, он будет перенаправлен на страницу Dashboard, где будет отображаться общая статистика по использованию музыкальной платформы. На этой странице manager сможет увидеть информацию о количестве зарегистрированных пользователей, количестве загруженных треков и жанров.

На странице Users manager может управлять пользователями. Он имеет возможность просмотреть список всех пользователей, а также добавить, удалить или отредактировать данные любого пользователя.

# Заключение

База данных является ключевым элементом любой современной организации, обеспечивая надежное хранение и управление информацией. В данной работе была поставлена задача разработки базы данных для музыкальной площадки с использованием технологии применения мультимедийных типов данных в СУБД PostgreSQL.

В процессе выполнения работы были использованы различные объекты, включая таблицы, триггеры и функции, чтобы обеспечить структурированное хранение данных и своевременный доступ к ним. В результате, цель работы была успешно достигнута, и база данных готова к использованию. Были разработаны роли для управления доступом к данным и обеспечения безопасности.

Тестирование базы данных было проведено при использовании большого объема данных, и результаты были положительными. Были реализованы процедуры для импорта и экспорта данных в формате JSON и XML, что обеспечило удобство использования и управления данными.

Одной из ключевых особенностей разработанной базы данных является технология хранения мультимедийных данных, что позволяет эффективно управлять медиа-файлами на площадке.

Таким образом, была успешно выполнена задача по разработке базы данных для музыкальной площадки на основе СУБД PostgreSQL. Разработанная база данных позволяет хранить и управлять большим объемом музыкальных данных, обеспечивает безопасный доступ к ним и предоставляет возможность импорта и экспорта данных в различных форматах. Кроме того, технология хранения мультимедийных данных позволяет эффективно управлять медиа-файлами на площадке.

# Список используемых источников

1. Spotify [Электронный ресурс] / Режим доступа: https://spotify.com – Дата доступа: 18.04.2023.
2. Apple Music [Электронный ресурс] / Режим доступа: https://applemusic.com/ – Дата доступа: 18.04.2023.
3. PostgreSQL Сайт о программировании [Электронный ресурс] / Режим доступа: https://postgrespro.ru/docs/postgresql.com – Дата доступа: 18.04.2023.
4. Postgresqltutorial.com [Электронный ресурс] / Режим доступа: https://www.postgresqltutorial.com/ – Дата доступа: 18.04.2023.
5. Stackoverflow.com [Электронный ресурс] / Режим доступа: https://stackoverflow.com – Дата доступа: 18.04.2023.

# Приложение А. Код создания таблиц

|  |
| --- |
| --------------------- Tablespace ---------------------------  CREATE TABLESPACE ts\_user LOCATION 'D:\DATA\DBS\ts\_user';  CREATE TABLESPACE ts\_track LOCATION 'D:\DATA\DBS\ts\_track';  CREATE TABLESPACE ts\_playlist LOCATION 'D:\DATA\DBS\ts\_playlist';  CREATE TABLESPACE ts\_library LOCATION 'D:\DATA\DBS\ts\_library';  /\*---------------------————————————————---------------------  -----------------------| CREATE TABLES |--------------------  ------------------------————————————————-------------------\*/  --------------------- Table Role ----------------------  CREATE TABLE Role (  role\_id SERIAL PRIMARY KEY,  role\_name VARCHAR(255) NOT NULL  ) TABLESPACE ts\_user;  --------------------- Table Users ----------------------  CREATE TABLE Users (  user\_id SERIAL PRIMARY KEY,  user\_name VARCHAR(255) NOT NULL,  user\_img BYTEA,  user\_email VARCHAR(255) UNIQUE NOT NULL,  user\_password VARCHAR(255) NOT NULL,  user\_date\_of\_birth DATE,  user\_role\_id INTEGER,  CONSTRAINT fk\_user\_role FOREIGN KEY (user\_role\_id) REFERENCES Role (role\_id)  ) TABLESPACE ts\_user;  --------------------- Table Genre ----------------------  CREATE TABLE Genre (  genre\_id SERIAL PRIMARY KEY,  genre\_name VARCHAR(255) NOT NULL  ) TABLESPACE ts\_track;  --------------------- Table Track ----------------------  CREATE TABLE Track (  track\_id SERIAL PRIMARY KEY,  track\_title VARCHAR(255) NOT NULL,  track\_date DATE NOT NULL,  track\_image BYTEA,  track\_content BYTEA NOT NULL,  genre\_id INTEGER NOT NULL,  CONSTRAINT fk\_track\_genre FOREIGN KEY (genre\_id) REFERENCES Genre (genre\_id)  ) TABLESPACE ts\_track;  --------------------- Table User\_track ----------------------  CREATE TABLE User\_Track (  user\_id INTEGER NOT NULL,  track\_id INTEGER NOT NULL,  PRIMARY KEY (user\_id, track\_id),  CONSTRAINT fk\_user\_track\_user FOREIGN KEY (user\_id) REFERENCES Users (user\_id),  CONSTRAINT fk\_user\_track\_track FOREIGN KEY (track\_id) REFERENCES Track (track\_id)  ) TABLESPACE ts\_track;  --------------------- Table Rating----------------------  CREATE TABLE Rating (  rating\_id SERIAL PRIMARY KEY,  user\_id INTEGER NOT NULL,  track\_id INTEGER NOT NULL,  rate INTEGER NOT NULL,  CONSTRAINT fk\_rating\_user FOREIGN KEY (user\_id) REFERENCES Users (user\_id),  CONSTRAINT fk\_rating\_track FOREIGN KEY (track\_id) REFERENCES Track (track\_id)  ) TABLESPACE ts\_track;  --------------------- Table Playlist ----------------------  CREATE TABLE Playlist (  playlist\_id SERIAL PRIMARY KEY,  user\_id INTEGER NOT NULL,  title VARCHAR(255) NOT NULL,  CONSTRAINT fk\_playlist\_user FOREIGN KEY (user\_id) REFERENCES Users (user\_id)  ) TABLESPACE ts\_playlist;  --------------------- Table Playlist\_tracks ----------------------  CREATE TABLE Playlist\_tracks (  id SERIAL PRIMARY KEY,  track\_id INTEGER NOT NULL,  playlist\_id INTEGER NOT NULL,  CONSTRAINT fk\_playlist\_tracks\_track FOREIGN KEY (track\_id) REFERENCES Track (track\_id),  CONSTRAINT fk\_playlist\_tracks\_playlist FOREIGN KEY (playlist\_id) REFERENCES Playlist (playlist\_id)  ) TABLESPACE ts\_playlist;  --------------------- Table Library\_user ----------------------  CREATE TABLE Library\_user (  id SERIAL PRIMARY KEY,  user\_id INTEGER NOT NULL,  track\_id INTEGER NOT NULL,  CONSTRAINT fk\_library\_track\_user FOREIGN KEY (user\_id) REFERENCES Users (user\_id),  CONSTRAINT fk\_library\_track\_track FOREIGN KEY (track\_id) REFERENCES Track (track\_id)  ) TABLESPACE ts\_library;  --------------------- End ---------------------- |

# Приложение Б. Код создания функций

|  |
| --- |
| CREATE OR REPLACE FUNCTION Register(  in\_user\_name VARCHAR(255),  in\_user\_email VARCHAR(255),  in\_user\_password TEXT,  in\_user\_date\_of\_birth DATE,  in\_user\_role\_id INTEGER,  in\_user\_img BYTEA  )  RETURNS INTEGER  LANGUAGE plpgsql  AS $$  DECLARE  encrypted\_password TEXT := encrypt\_password(in\_user\_password);  user\_id INTEGER;  BEGIN  INSERT INTO Users(user\_name, user\_email, user\_password, user\_date\_of\_birth, user\_role\_id, user\_img)  VALUES (in\_user\_name, in\_user\_email, encrypted\_password, in\_user\_date\_of\_birth, in\_user\_role\_id, in\_user\_img)  RETURNING Users.user\_id INTO user\_id;  RETURN user\_id;  END;  $$;  SELECT Register('admin', 'admin', 'admin', '2004-10-09', 1, '\x1234567890ABCDEF');  SELECT Register('ans', 'ans', 'ans', '1995-01-01', 2, '\x1234567890ABCDEF');  --------------------- Login user---------------------  CREATE OR REPLACE FUNCTION Login(in\_user\_email VARCHAR(255), in\_user\_password TEXT)  RETURNS BOOLEAN  AS $$  DECLARE  stored\_password TEXT;  BEGIN  SELECT user\_password INTO stored\_password FROM Users WHERE user\_email = in\_user\_email;    IF stored\_password IS NULL THEN  RETURN FALSE;  END IF;  RETURN stored\_password = encrypt\_password(in\_user\_password);  END;  $$ LANGUAGE plpgsql;  SELECT Login('admin', 'admin');  select \* from users  -------------------- Add Track --------------------  CREATE OR REPLACE FUNCTION AddTrack(  in\_track\_title VARCHAR(255),  in\_track\_date DATE,  in\_track\_image BYTEA,  in\_track\_content BYTEA,  in\_genre\_id INTEGER  )  RETURNS INTEGER  LANGUAGE plpgsql  AS $$  DECLARE  track\_id INTEGER;  BEGIN  INSERT INTO Track(track\_title, track\_date, track\_image, track\_content, genre\_id)  VALUES (in\_track\_title, in\_track\_date, in\_track\_image, in\_track\_content, in\_genre\_id)  RETURNING Track.track\_id INTO track\_id;    RETURN track\_id;  END;  $$;  select AddTrack('Название трека', '2024-05-08', NULL, 'содержимое трека', 1);  CALL AddTrack('Название трека', '2024-05-08', NULL, 'содержимое трека', 1);  call AddUserTrack(3,1)  -------------------- Add Track user --------------------  CREATE OR REPLACE PROCEDURE AddUserTrack(  in\_user\_id INTEGER,  in\_track\_id INTEGER  )  LANGUAGE plpgsql  AS $$  BEGIN  INSERT INTO User\_Track(user\_id, track\_id)  VALUES (in\_user\_id, in\_track\_id);  END;  $$;  select \* from user\_track;  -------------------- Create playlist --------------------  CREATE OR REPLACE FUNCTION CreatePlaylist(  in\_user\_id INTEGER,  in\_title VARCHAR(255)  )  RETURNS INTEGER  LANGUAGE plpgsql  AS $$  DECLARE  playlist\_id INTEGER;  BEGIN  INSERT INTO Playlist(user\_id, title)  VALUES (in\_user\_id, in\_title)  RETURNING Playlist.playlist\_id INTO playlist\_id;    RETURN playlist\_id;  END;  $$;  select CreatePlaylist(43,'mu list')  select createPlaylist(1, 'My Playlist');  select \* from Playlist; |

# Приложение В. Выполнение процедур

|  |
| --- |
| CREATE OR REPLACE PROCEDURE AddUserTrack(  in\_user\_id INTEGER,  in\_track\_id INTEGER  )  LANGUAGE plpgsql  AS $$  BEGIN  INSERT INTO User\_Track(user\_id, track\_id)  VALUES (in\_user\_id, in\_track\_id);  END;  $$;  select \* from user\_track  -------------------- Add rating --------------------  CREATE OR REPLACE PROCEDURE add\_rating(  in\_user\_id INTEGER,  in\_track\_id INTEGER,  in\_rate INTEGER  )  LANGUAGE plpgsql  AS $$  BEGIN    IF NOT EXISTS (SELECT 1 FROM Users WHERE user\_id = in\_user\_id) THEN  RAISE EXCEPTION 'User with id % does not exist', in\_user\_id;  END IF;    IF NOT EXISTS (SELECT 1 FROM Track WHERE track\_id = in\_track\_id) THEN  RAISE EXCEPTION 'Track with id % does not exist', in\_track\_id;  END IF;    INSERT INTO Rating(user\_id, track\_id, rate)  VALUES (in\_user\_id, in\_track\_id, in\_rate);  END;  $$;  CALL add\_rating(1, 4, 5);  delete from Rating;  select \* from Rating;$$ LANGUAGE plpgsql;  -------------------- Add track in playlist --------------------  CREATE OR REPLACE PROCEDURE add\_track\_to\_playlist(  in\_track\_id INTEGER,  in\_playlist\_id INTEGER  )  LANGUAGE plpgsql  AS $$  BEGIN  INSERT INTO Playlist\_tracks(track\_id, playlist\_id)  VALUES (in\_track\_id, in\_playlist\_id);  END;  $$;  CALL add\_track\_to\_playlist(2,1);  select \* from Playlist\_tracks;  -------------------- Add track in library --------------------  CREATE OR REPLACE PROCEDURE add\_track\_to\_library(  in\_user\_id INTEGER,  in\_track\_id INTEGER  )  LANGUAGE plpgsql  AS $$  BEGIN  INSERT INTO Library\_user(user\_id, track\_id) VALUES (in\_user\_id, in\_track\_id);    END;  $$;  call add\_track\_to\_library(1,1)  drop PROCEDURE add\_track\_to\_library  select \* from Library\_user; |

# Приложение Г. Роли клиентов

|  |
| --- |
| CREATE ROLE PROGRAMMER\_ROLE;  CREATE ROLE USER\_ROLE;  CREATE ROLE MANAGER\_ROLE;  -------------- PROGRAMMER\_ROLE ------------------  GRANT ALL PRIVILEGES ON DATABASE "postgres" TO PROGRAMMER\_ROLE;  GRANT CONNECT ON DATABASE "postgres" TO PROGRAMMER\_ROLE;  GRANT ALL ON TABLESPACE TS\_USER TO PROGRAMMER\_ROLE;  GRANT ALL ON TABLESPACE TS\_TRACK TO PROGRAMMER\_ROLE;  GRANT ALL ON TABLESPACE TS\_PLAYLIST TO PROGRAMMER\_ROLE;  GRANT ALL ON TABLESPACE TS\_LIBRARY TO PROGRAMMER\_ROLE;  -------------- USER\_ROLE ------------------  GRANT EXECUTE ON FUNCTION REGISTER TO USER\_ROLE;  GRANT EXECUTE ON FUNCTION LOGIN TO USER\_ROLE;  GRANT EXECUTE ON FUNCTION ADDTRACK TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE ADD\_RATING TO USER\_ROLE;  GRANT EXECUTE ON FUNCTION CREATEPLAYLIST TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE ADD\_TRACK\_TO\_PLAYLIST TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE ADD\_TRACK\_TO\_LIBRARY TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE ADD\_TRACK\_TO\_LIBRARY TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_USER TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_USER\_PASSWORD TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_TRACK TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_PLAYLIST TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE DELETE\_TRACK TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE REMOVE\_TRACK\_FROM\_PLAYLIST TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE DELETE\_PLAYLIST TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE DELETE\_TRACK\_FROM\_LIBRARY TO USER\_ROLE;  GRANT EXECUTE ON FUNCTION GETUSERBYID TO USER\_ROLE;  GRANT EXECUTE ON FUNCTION SEARCH\_TRACK\_BY\_TITLE\_OR\_USER\_NAME TO USER\_ROLE;  GRANT SELECT, INSERT, UPDATE ON TABLE USERS TO USER\_ROLE;  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE TRACK TO USER\_ROLE;  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE RATING TO USER\_ROLE;  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE USER\_TRACK TO USER\_ROLE;  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE PLAYLIST TO USER\_ROLE;  GRANT SELECT, INSERT, UPDATE ON TABLE LIBRARY\_USER TO USER\_ROLE;  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE PLAYLIST\_TRACKS TO USER\_ROLE;  GRANT USAGE, SELECT ON SEQUENCE USERS\_USER\_ID\_SEQ TO USER\_ROLE;  GRANT USAGE, SELECT ON SEQUENCE TRACK\_TRACK\_ID\_SEQ TO USER\_ROLE;  GRANT USAGE, SELECT ON SEQUENCE RATING\_RATING\_ID\_SEQ TO USER\_ROLE;  GRANT USAGE, SELECT ON SEQUENCE PLAYLIST\_PLAYLIST\_ID\_SEQ TO USER\_ROLE;  GRANT USAGE, SELECT ON SEQUENCE LIBRARY\_USER\_ID\_SEQ TO USER\_ROLE;  -------------- MANAGER\_ROLE ------------------  GRANT EXECUTE ON FUNCTION REGISTER TO MANAGER\_ROLE;  GRANT EXECUTE ON FUNCTION GETUSERBYID TO MANAGER\_ROLE;  GRANT EXECUTE ON FUNCTION SEARCH\_TRACK\_BY\_TITLE\_OR\_USER\_NAME TO MANAGER\_ROLE;  GRANT EXECUTE ON FUNCTION ADDTRACK TO MANAGER\_ROLE;  GRANT EXECUTE ON FUNCTION GETTRACKS TO MANAGER\_ROLE;  GRANT EXECUTE ON FUNCTION GETUSERS TO MANAGER\_ROLE;  GRANT EXECUTE ON FUNCTION GETTRACKBYID TO MANAGER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_USER TO MANAGER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_USER\_PASSWORD TO MANAGER\_ROLE;  GRANT EXECUTE ON PROCEDURE DELETE\_TRACK TO MANAGER\_ROLE;  GRANT EXECUTE ON PROCEDURE DELETE\_USER TO MANAGER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_TRACK TO MANAGER\_ROLE;  GRANT SELECT, INSERT, UPDATE ON TABLE USERS TO MANAGER\_ROLE;  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE TRACK TO MANAGER\_ROLE;  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE USER\_TRACK TO MANAGER\_ROLE;  GRANT USAGE, SELECT ON SEQUENCE USERS\_USER\_ID\_SEQ TO MANAGER\_ROLE;  GRANT USAGE, SELECT ON SEQUENCE TRACK\_TRACK\_ID\_SEQ TO MANAGER\_ROLE; |