Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет   
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина «Модели данных и системы управления базами данных»

|  |  |
| --- | --- |
|  | «К ЗАЩИТЕ ДОПУСТИТЬ» |
|  | Руководитель курсового проекта  ассистент кафедры информатики  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_В.С.Плиска |
|  | \_\_\_.\_\_\_\_.2023 |

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту

на тему

**БАЗА ДАННЫХ МУЗЫКАЛЬНОГО СЕРВИСА**

БГУИР КП 1-40 04 01

|  |  |
| --- | --- |
|  | Выполнил студент группы 053502  Бокун Артем Геннадьевич  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись студента) |
|  | Курсовой проект представлен на проверку \_\_\_.\_\_\_\_.2023  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись студента) |

Минск 2023

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 4](#_Toc1)

[1 Обзор аналогов 5](#_Toc2)

[1.1 Spotify 5](#_Toc3)

[1.2 SoundCloud 8](#_Toc4)

[1.3 Яндекс Музыка 9](#_Toc5)

[1.4 Формирование функциональных требований 12](#_Toc6)

[2 Обзор технологий и изучение литературных источников 13](#_Toc7)

[2.1 Реляционные базы данных 13](#_Toc8)

[2.2 Общие сведения о реляционных базах данных 13](#_Toc9)

[2.3 Нормализация базы данных 15](#_Toc10)

[2.4 Язык SQL 21](#_Toc19)

[2.5 Типы данных в языке SQL 21](#_Toc20)

[2.6 Хранимый код в языке SQL и его диалектах 24](#_Toc27)

[2.7 Индексы в языке SQL 26](#_Toc30)

[2.8 Нереляционные базы данных 27](#_Toc31)

[2.9 Документо-ориентированные базы данных 27](#_Toc32)

[2.10 MongoDB 28](#_Toc33)

[2.11 Графовые базы данных 29](#_Toc34)

[2.12 Neo4j 29](#_Toc35)

[2.13 Язык Cypher 30](#_Toc36)

[2.14 Выбор базы данных 30](#_Toc37)

[3 Проектирование базы данных 31](#_Toc38)

[3.1 Разработка структуры базы данных 31](#_Toc39)

[3.2 Создание ERD диаграмм 34](#_Toc40)

[3.3 Создание IDEF1X диаграммы 35](#_Toc41)

[4 Разработка физической базы данных 36](#_Toc42)

[4.1 Создание функций 36](#_Toc43)

[4.2 Создание триггеров 37](#_Toc44)

[4.3 Создание индексов 41](#_Toc45)

[5 Тестирование базы данных 42](#_Toc46)

[5.1 Проверка триггеров 42](#_Toc45)

[5.2 Проверка функций 44](#_Toc45)

[Заключение 46](#_Toc47)

[Список использованной литературы 47](#_Toc48)

[Приложение А (обязательное) листинг кода 48](#_Toc49)

#### ВВЕДЕНИЕ

Рынок стриминговых музыкальных сервисов на сегодняшний день является одним из важнейших медиа-рынков. По данным агентства FutureSource Consulting, в 2023 году число подписок на музыкальные стриминговые сервисы превзошло 425 млн, и 90% всех денег, потраченных слушателями на музыку, придутся на стриминг. В 2024 году число подписок на музыкальные сервисы может вырасти на 15%. Согласно исследованиям Mordor Intelligence Private Limited в период с 2018 по 2028 год ожидается среднегодовой прирост пользователей в размере 13%.

Столь серьезные темпы роста становятся весомой причиной для включения многих крупных игроков IT-сектора выпускать свои музыкальные сервисы. Такие есть например YouTube (YouTube Music), Google Play (Google Play Музыка), Apple (Apple Music), Amazon (Amazon Music) многих других. Как видно рынок настолько перспективный и крупный, что корпорации не боятся иметь у себя несколько конкурирующих стриминговых сервисов, как например Google, имеющая одновременно и YouTube Music, и Google Play Музыка.

Не стоит забывать и о компаниях, которые полностью специализируются на музыкальном стриминге. Spotify на сегодняшний день контролирует наибольшую часть музыкального рынка, рядом с ним находятся компании вроде Deezer, SoundCloud и Tidal. Как видно, конкуренция между игроками данного рынка довольно серьезная, появиться новому сервису без внушительного начального капитала в таких условиях крайне сложно.

Что касается рынка стран СНГ, то здесь долгое время крупных мировых игроков не было вовсе, сейчас же они появились, но у большого количества граждан стран СНГ нет возможности оплатить подписку для пользования иностранными сервисами. Такая ситуация позволила сформироваться локальному рынку стриминговых музыкальных сервисов, главным из которых сегодня является Яндекс Музыка. Кроме Яндекс Музыки существуют проекты VK (VK Музыка), Сбера (СберЗвук) и МТС (МТС Music). Однако Яндекс Музыка на сегодняшний день является абсолютно доминирующим агентом, по данным MIDiA Research данный сервис занимает 63.3% российского рынка.

На сегодняшний день рынок стран СНГ является развивающимся и не столь требовательным к начальным данным выходящего на него агента, что позволяет сделать вывод об обоснованности создания более качественного музыкального сервиса, ориентированного на страны СНГ.

В рамках настоящей работы, будет рассмотрен аспект создания базы данных для такого проекта, описаны технологии и методики проектирования подобного рода систем, а так же разработан элементарный пример такой базы данных. Кроме этого будут рассмотрены различные виды баз данных и уровень их пригодности для выполнения разного рода задач в рамках одной системы.

#### 1 ОБЗОР АНАЛОГОВ

Для понимания на что следует ориентироваться при разработке музыкального сервиса следует обратить внимание на успешных представителей данного рынка. В этой главе для рассмотрения возьмем Spotify, как наиболее популярный стриминговый музыкальный сервис в мире, Apple Music, как наиболее влиятельный в свое время сервис и Яндекс Музыку, как наиболее популярный сервис на целевом для разрабатываемой платформы рынке.

#### 1.1 Spotify

Сервис Spotify появился в Швеции в 2006 году. За первые несколько лет своего существования эта платформа превратилась из локального стартапа с сомнительной бизнес-моделью в главный музыкальный сервис Европы и начала представлять опасность для Apple с их ITunes. На рисунке 1.1 представлена главная страница приложения Spotify для настольных компьютеров.

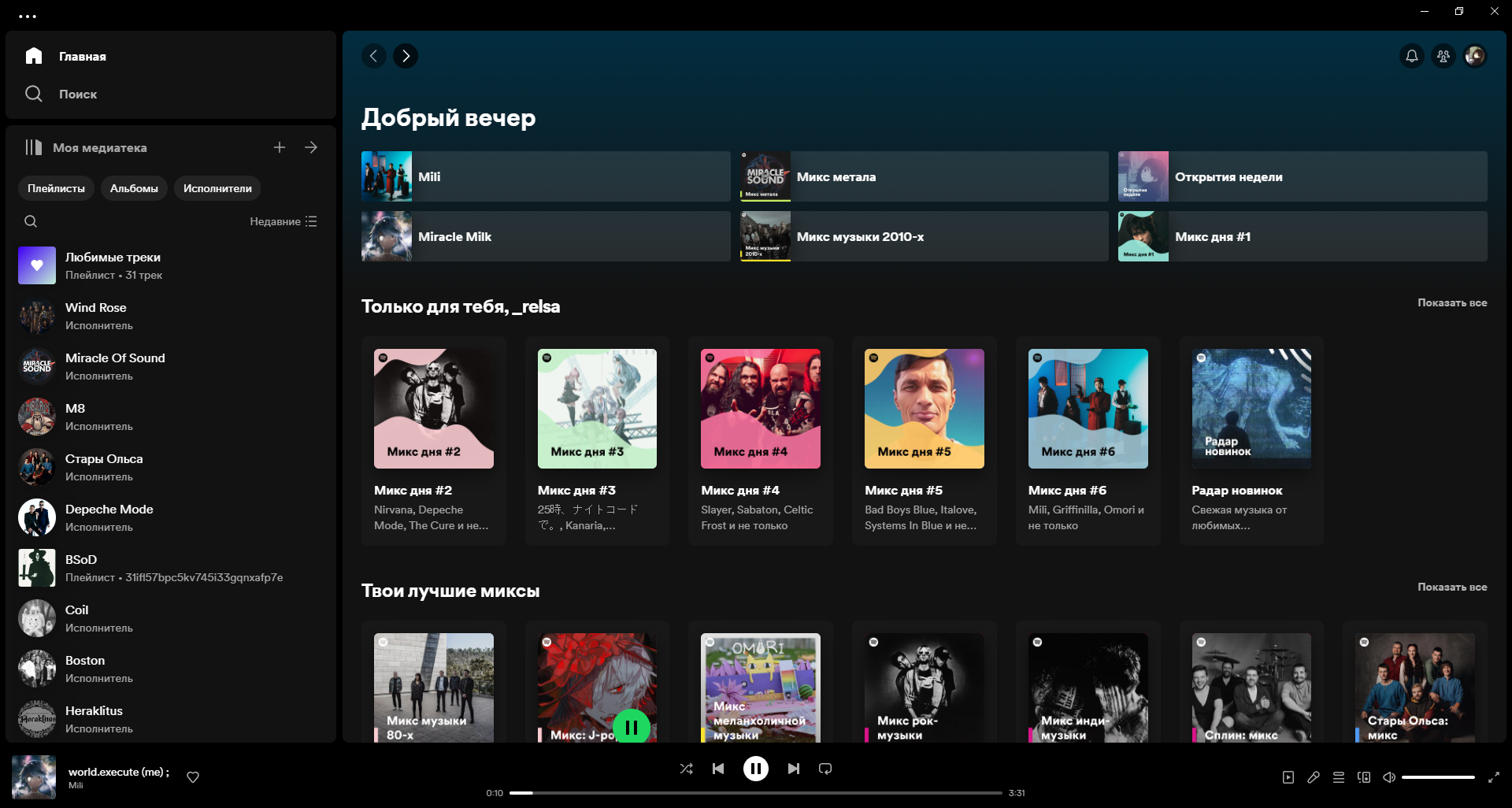


Рисунок 1.1 — Главная страница приложения Spotify для настольных компьютеров

Создатели Spotify в проектировании своего сервиса ориентировались на механизмы, которыми пользовались пираты для быстрого обмена большим количеством контента. Даниэль Эк хотел создать легальный аналог пиратских сервисов, которые на тот момент были гораздо удобнее их легальных аналогов. Той же стратегии придерживались и создатели ITunes [1].

Всю свою историю Spotify делал ставку на фримиум модель, которая считалась сомнительной, как крупными IT-компаниями, так и корпорациями, которые владели лицензионными правами на музыку. Spotify же придерживался ее с момента запуска. По оценкам Даниэля Эка в начале работы сервиса подписку должны были оформить лишь 2-15 процентов пользователей, остальные же в целом могли быть удовлетворены бесплатной версией [1].

Звукозаписывающие компании не видели перспектив в буквально бесплатном распространении музыки и поэтому ставили максимально жесткие условия для Spotify в области лицензирования контента. Однако сервис выдержал и сегодня фримиум стала основной моделью дистрибуции цифрового контента. Популяризировал ее именно Spotify. Базовая страница альбома, плейлиста или радио Spotify представлена на рисунке 1.2.

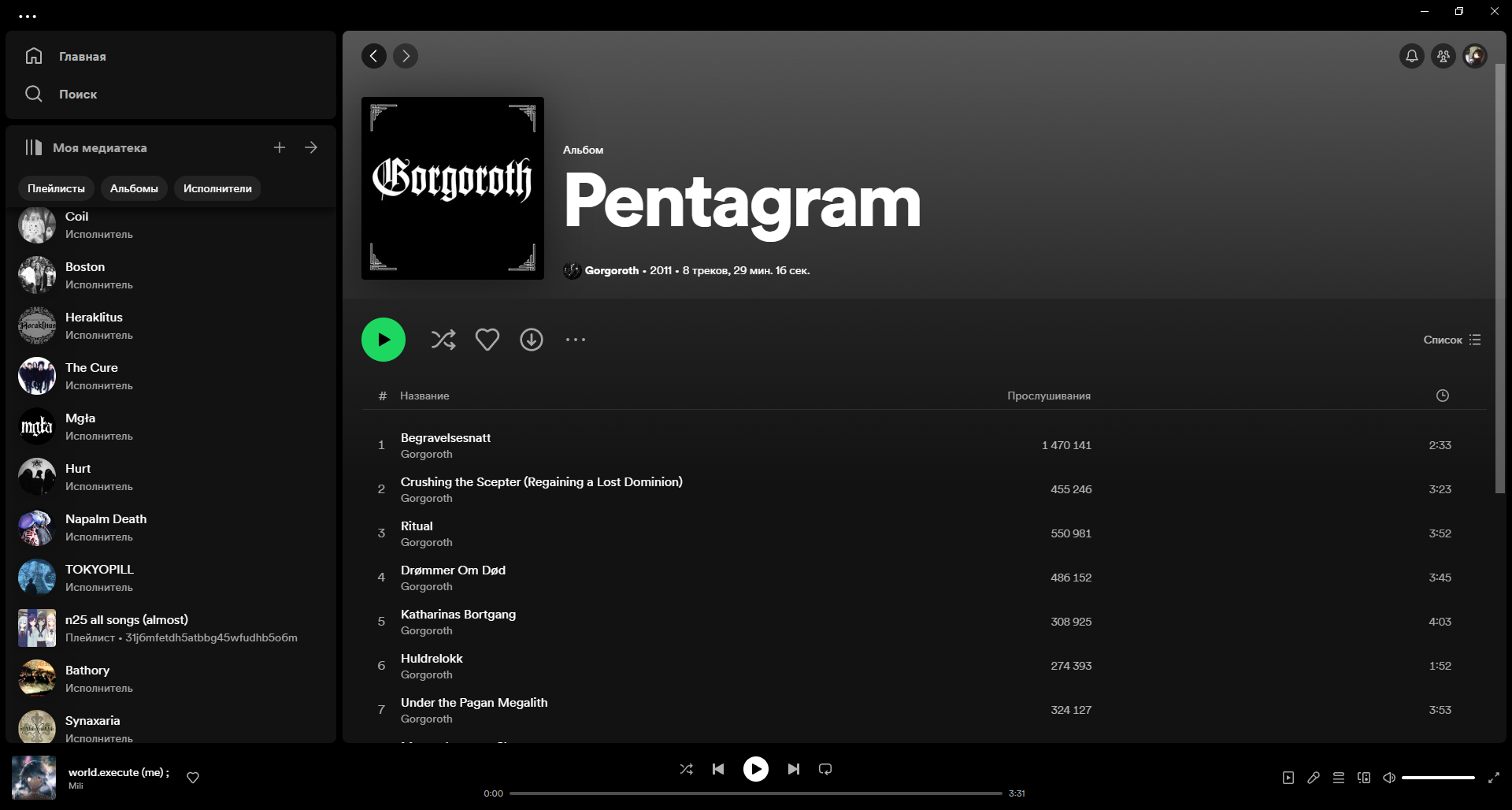


Рисунок 1.2 — Базовая страница альбома, плейлиста или радио Spotify

Когда Spotify выходила на рынок Северной Америки, то столкнулась с серьезнейшим противодействием со стороны Apple. Сервису нужны были новые идеи для победы над конкурентом, итогом поисков стало развитие направлений анализа больших данных и искусственного интеллекта. Работа с большими данными в 2014 году была магистральной для Spotify, разработчики хотели чтобы сервис подбирал музыку не только по предпочтениям пользователя, но и согласно текущему его состоянию. То есть, если пользователь спешит на работу или занимается спортом, то ему должна была предлагаться музыка одного характера, а если собирается отходить ко сну, то другого. Проект получил название Moments [1].

Подразделение исследований искусственного интеллекта занималось исследованием более точных рекомендаций на основе музыкальных вкусов пользователя. Обучаемой модели предоставили данные о более чем 1.5 миллиардов пользовательских плейлистов. Обучившись на них модель смогла самостоятельно составить несколько музыкальных плейлистов для сотрудников компании. Резульат получился настолько впечатляющим, что Меттью Огле позволил внедрить технологию в сервис. Теперь каждую неделю пользователю предлагается плейлист из 30 музыкальных композиций, которые алгоритм посчитал подходящими ему. Функцию назвали «Открытия недели». На рисунке 1.3 представлена страница плейлиста «Открытия недели».

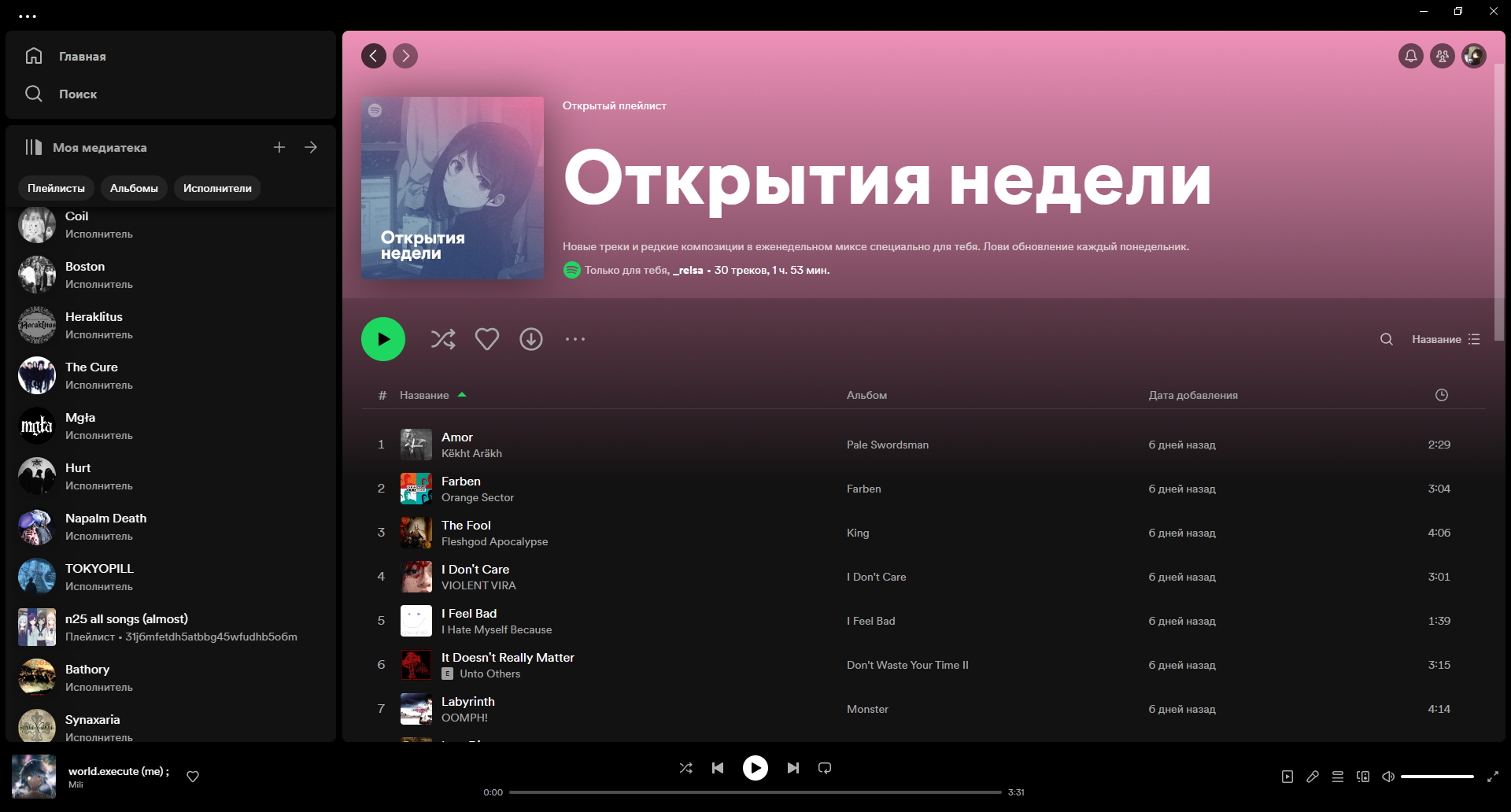


Рисунок 1.3 — Страница плейлиста «Открытия недели»

Сегодня Spotify является ведущим сервисом на рынке музыкального стриминга. Spotify имеет хорошую библиотеку музыки, неплохие алгоритмы и большую аудиторию пользователей. Из плюсов платформы можно выявить.

* Кросспалтформенность. Spotify представлен на всех видах устройств: от мобильных телефонов и до автомобильных компьютеров.
* Персонализированные рекомендации: Spotify использует алгоритмы машинного обучения, чтобы предлагать пользователям песни и подкасты, которые им могут понравиться.
* Социальные функции: Spotify позволяет пользователям создавать плейлисты и делиться ими с друзьями и семьей.
* Бесплатный тариф: Spotify предоставляет бесплатный тариф, который позволяет пользователям слушать музыку с рекламой.
* Большая библиотека: Spotify предоставляет доступ к миллионам песен и подкастов, что делает его одним из самых больших музыкальных сервисов в мире.

Из минусов можно выделить.

* Качество звука. Качество звука на Spotify не такое высокое, как на других сервисах, таких как Apple Music. Некоторые пользователи заметили, что звук на Spotify звучит тише и менее детализировано, чем на других сервисах.
* Отсутствие понятного раздела с новой музыкой. Spotify не имеет явного раздела для новых альбомов и треков, что может быть неудобно для пользователей.
* Отсутствие светлой темы. Spotify не предоставляет светлую тему для пользователей, которые предпочитают светлый интерфейс.
* Рекомендации иногда теряются. Рекомендации на главной странице Spotify могут быть неочевидными и теряться среди других элементов.

#### 1.2 SoundCloud

Еще одной музыкальной платформой, появившейся в Швеции в 2006 году стал SoundCloud. В отличии от Spotify SoundCloud изначально ориентировался на модель социальных сетей. Главным конкурентом этого сервиса в области распространения музыки считался MySpace. Интерфейс главной страницы SoundCloud представлен на рисунке 1.4.

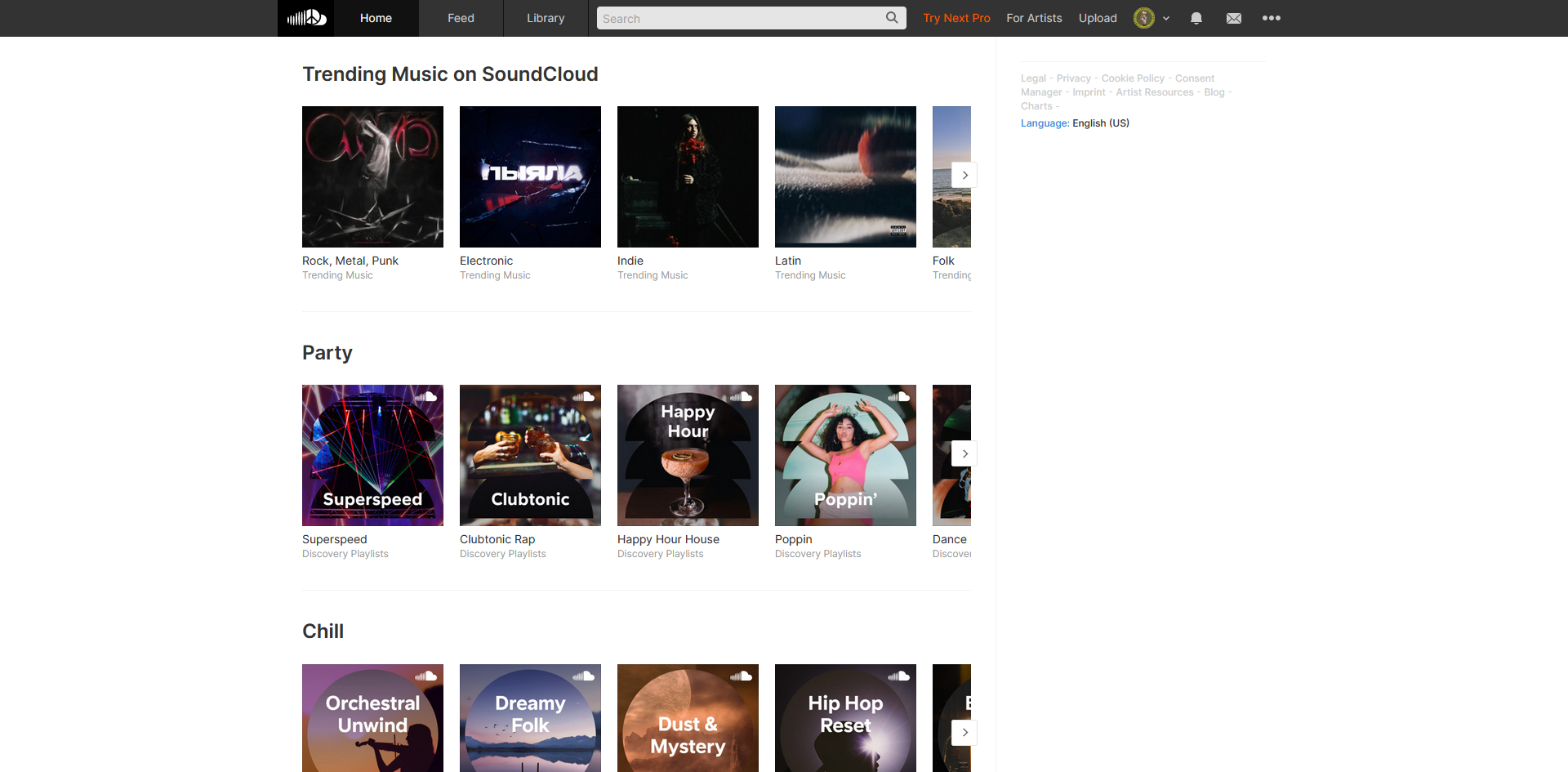


Рисунок 1.4 — Интерфейс главной страницы SoundCloud

Поскольку SoundCloud изначально предполагался быть социальной сетью, где музыканты обменивались бы друг с другом своими произведениями, а лишь потом стал стриминговым музыкальным сервисом, в нем сохранилось много элементов присущих социальным сетям. Пользователи могут делать репосты композиций к себе на страницу, оставлять лайки, которые будут видны другим пользователям и писать комментарии, причем не только к всей песне целиком, но и к отдельным ее моментам. На рисунке 1.5 представлен пример комментария к определенному моменту в песне.



Рисунок 1.5 — Пример комментария к определенному моменту в песне

Другим важным аспектом архитектуры и функциональности SoundCloud является легкая интеграция с другими сервисами. SoundCloud представляет интерфейсы для запуска песен на других сайтах, а так же публичные API для работы с аудио-файлами.

Помимо этого присутствуют и стандартные функции вроде составления плейлистов и сохранения понравившихся песен.

Из преимуществ сервиса можно отметить:

* огромная библиотека довольно нишевых произведений;
* простой и удобный интерфейс;
* встроенные инструменты продвижения и обратной связи;
* бесшовная интеграция с другими платформами;
* большое и вовлеченное сообщество пользователей и артистов.

К минусам же можно отнести:

* ограниченная аналитика по сравнению с аналогичными сервисами;
* бесплатные аккаунты имеют ограничения по загрузке;
* низкое качество звука для бесплатных пользователей;
* много рекламы и спама;
* нестабильность и сбои сервиса.

#### 1.3 Яндекс Музыка

Яндекс Музыка является полным аналогом вышеприведенных сервисов для постсоветского рынка. Запуск сервиса прошел в 2010 году, в Яндексе оценили перспективный рынок музыкального стриминга и приняли решение развивать работающую бизнес-модель Spotify и ITunes для локальных пользователей.

Платформа обладает всем стандартным функционалом музыкального стримингового сервиса: составление плейлистов, персонализированные рекомендации, списки избранных песен и некоторыми другими. Интерфейс плейлиста Яндекс Музыки представлен на рисунке 1.6.

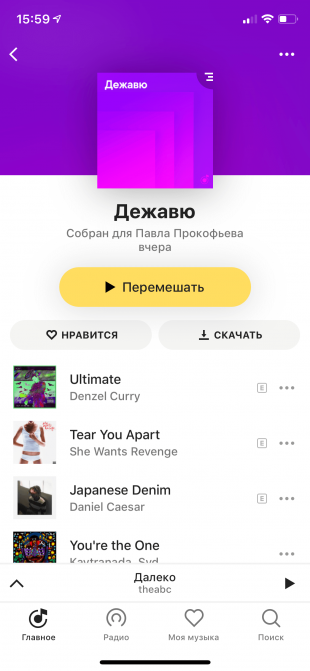


Рисунок 1.6 — Интерфейс плейлиста Яндекс Музыки

Стоит отметь, что Яндекс Музыка ориентируется не только на Spotify в своей модели, но и на SoundClound, так например плеер Яндекс Музыки можно легко интегрировать в сторонние приложения.

Кроме этого интересным нововведением, прямого аналога которых у других сервисов нет — это функция «Моя волна». Данная функция похожа на функцию «Открытия недели» в Spotify. Однако в Spotify она реализована в качестве ограниченного плейлиста и подновляется раз в неделю, а в Яндекс Музыке в качестве радиостанции. На рисунке 1.7 изображен интерфейс функции «Моя волна».

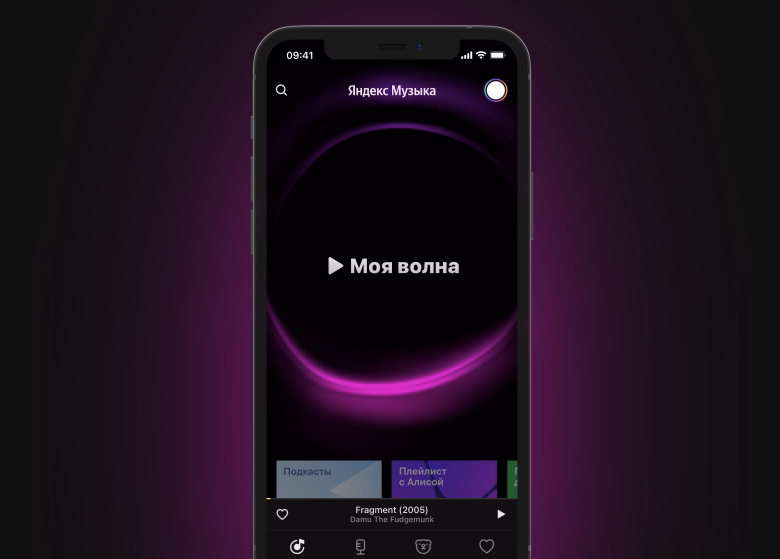


Рисунок 1.7 — Интерфейс функции «Моя волна»

Подводя итог, у Яндекс Музыки можно выделить следующие плюсы:

* обширный каталог локальных исполнителей;
* кураторские плейлисты от известных медиа, радиостанций и экспертов в разных жанрах;
* возможность распознавать треки, которые играют вокруг, с помощью встроенного аналога Shazam.

В качестве минусов сервиса можно обозначить:

* оформление подписки производится только через сайт, а не через приложение;
* низкокачественная модерация треков, которая иногда пропускает некачественные или неподходящие записи;
* недружественный интерфейс в веб-версии, который неудобен для навигации и поиска;
* отсутствие расширенной статистики прослушиваний, которая позволила бы узнать свои музыкальные предпочтения и привычки.

#### 1.4 Формирование функциональных требований

Изучив ведущие мировые платформы музыкального стриминга можно сформировать следующие требования к базе данных разрабатываемого проекта:

1. Система должна собирать и систематизировать статистические данные о пользователях.
2. Сервис должен реализовывать в себе социальные функции: комментирование записей, выставление оценок, некоторые социальные события.
3. На платформе должна присутствовать возможность стандартного взаимодействия с композициями: сохранение, составление плейлистов.
4. Сервис должен хранить метаданные о композициях, исполнителях и радиостанциях.
5. Сервис должен предоставлять пользователю возможность оформлять подписку различными способами и обеспечить высокий уровень защиты персональных платежных данных.
6. Сервис должен хранить информацию о событиях безопасности.
7. База данных должна быть легко расширяемой и иметь возможность работы в кластере.
8. База данных должна показывать высокую производительность при работе с большим объемом данных.
9. База данных должна быть отказоустойчивой и надежной, иметь развитые механизмы резервного копирования и репликации.
10. База данных должна быть легко мониторить и анализировать, чтобы быстро выявлять проблемы и устранять их.

#### 2 ОБЗОР ТЕХНОЛОГИЙ И ИЗУЧЕНИЕ ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ

Разработка базы данных для столь сложной системы требует изучения и правильного применения современных технологий хранения данных и грамотного комбинирования различных видов баз данных. В этой главе будут рассмотрены основные сведения, касающиеся баз данных, их видов и подходов к проектированию.

#### 2.1 Реляционные базы данных

Сначала необходимо определиться с основным понятием, а именно базой данных. Обычно данные, хранимые в базах данных называют перманентными. Данная характеристика позволяет четко отделить данные, хранящиеся в базах данных, от других данных, которые может использовать система в своей работе и хранение которых носит временный характер. К таким данным относятся например промежуточные результаты,управляющие операторы, рабочие очереди, программные управляющие блоки и некоторые другие виды данных, которые не предполагают постоянного хранения. Точнее говоря, можно утверждать, что данные в базе являются перманентными, так как, после их внесения в базу, удаление может быть возможно только после прямого запроса к этой базе, но не в результате какого-то побочного эффекта от выполнения какой-либо программы.

Итак, база данных — это некоторый набор перманентных (постоянно хранимых) данных, используемых прикладными программными системами какого-либо предприятия [2].

Пользователь имеет возможность проводить с данными в базе различные операции:

* операции вставки;
* операции обновления;
* операции удаления;
* операции выборки по различным критериям.

Все эти операции в реляционных базах данных выполняются с помощью специального языка составления запросов — SQL.

В данной работе будут рассмотрены три вида баз данных: реляционные, нереляционные и графовые (как подвид нереляционных).

#### 2.2 Общие сведения о реляционных базах данных

В теории баз данных важную роль играют понятия сущностей и связей. Как правило, сущностями называют любой различимый объект, который может быть представлен в базе данных [2].

Важной характеристикой любой сущности является наличие у нее некоторых свойств, которые необходимы для конкретизации объекта хранимых данных. Некоторые сущности имеют своей целью именно хранение данных о конкретизируемом объекте, а некоторые носят исключительно утилитарный характер и нужны для поддержания правильного функционирования базы данных. Примером свойства второго типа является первичный ключ и внешний ключ.

Первичный ключ — это уникальный идентификатор записи в таблице. Первичный ключ, как следует из определения, всегда должен быть уникальным и никогда не должен быть нулевым. В качестве первичного ключа может служить примитивная последовательность чисел, генерируемая через автоинкремент первичного ключа предыдущей записи, а может и более продвинутая технология вроде генерации UUID [3].

Внешний ключ — это поле в таблице, значение которого соответствует первичному ключу в другой таблице. Пара внешнего и первичного ключа служат для создания связей внутри базы данных [3].

Связями же принято считать взаимоотношения сущностей, которые формализуются через ссылки некоторых полей неключевых полей одной сущности на ключевые поля другой. Как правило такие связи носят бинарный характер, то есть прослеживаются в любом направлении.

Реляционные базы данных в корне своей архитектуры имеют формальную теорию, называемую реляционной моделью данных. Кроме всего прочего, в такой системе выполняются как минимум три условия:

1. Структурный аспект. Данные в базе воспринимаются пользователем, как таблицы (и никак иначе).
2. Аспект целостности. Эти таблицы отвечают определенным условиям целостности (которые будут рассматриваться в конце раздела).
3. Аспект обработки. В распоряжении пользователя имеются операторы манипулирования таблицами (например, предназначенные для поиска данных), которые генерируют новые таблицы на основании уже имеющихся и среди которых есть, по крайней мере, операторы сокращения (restrict), проекции (project) и объединения (join) [2].

Во введении к главе уже были описаны некоторые операции, которые пользователь может совершать с объектами данных (строками таблицы). В аспекте обработки же были указаны операции, совершаемые над таблицами. Рассмотрим их подробнее:

1. Операция сокращения извлекает указанные строки из таблицы. Следует отметить, что операцию сокращения иногда называют выборкой, однако соответствующий оператор не полностью соответствует оператору SELECT языка SQL.
2. Операция проекции предназначена для извлечения определенных столбцов из таблицы.
3. Операция соединения предназначена для получения комбинации двух таблиц на основе общих значений в общих столбцах. Иными словами, при выполнении данной операции строки некоторой таблицы N, объединяются со строками таблицы D по принципу наличия у них одинаковых значений некоторых полей (как правило строка таблицы N через внешний ключ ссылается на строку таблицы D) [2].

Необходимо также отметить одну важную особенность приведенных выше операций над таблицами, результатом их выполнения всегда будет новая таблица, над которой можно будет провести те же операции. Это свойство в реляционной модели называется свойством замкнутости. Оно играет очень важную роль во всей концепции реляционных баз данных и языка SQL. Свойство замкнутости позволяет создавать сложные конструкции обработки и выборки данных, без него эффективная работа SQL была бы невозможна.

#### 2.3 Нормализация базы данных

Одной из основных проблем с которой могут столкнуться разработчики при проектировании базы данных это избыточность данных в отношениях. В общем виде эту задачу можно представить как ситуацию при которой в разных отношениях дублируются одни и те же данные или же поля неправильно соотнесены с сущностями, которые они должны описывать (как правило, такие проблемы возникают в комплексе).

Предметом нормализации, в сущности, является создание формальных правил для имплементации некоторых базовых принципов проектирования баз данных (таких как принцип «по одному факту в одном месте») в разрабатываемые проекты. Для приведения базы данных в состояние некоторой нормальной формы используется процедура нормализации. Ее можно охарактеризовать, как последовательное приведение заданного набора переменных отношения к некоторой все более желательной форме. Следует отметить, что эта процедура обратима, то есть всегда можно использовать ее результат (например, множество переменных отношения, находящихся в ЗНФ) для обратного преобразования (в исходную переменную отношения, находящуюся в 2НФ). Возможность выполнения обратного преобразования является весьма важной характеристикой, поскольку это означает, что в процессе нормализации сохраняется информация (не происходит ее потеря) [2].

На сегодняшний день существуют шесть номерных нормальных форм, нормальная форма Бойса-Кодда и доменно-ключевая нормальная форма. Рассматривать их следует по порядку, так как каждая следующая нормальная форма имеет своим условием нахождение базы данных в предыдущей нормальной форме.

#### 2.3.1 Первая нормальная форма (1НФ)

Первая нормальная форма является основной нормальной формой для всех баз данных, построенных на реляционной модели данных. Суть ее можно описать следующим образом:

Переменная отношения находится в 1НФ тогда и только тогда, когда в любом допустимом значении этой переменной отношения каждый ее кортеж содержит только одно значение для каждого из атрибутов [2].

Иными словами, здесь утверждается, что любое отношение по умолчанию находится в первой нормальной форме, что верно. Однако также верно, что отношение, которое находится только в первой нормальной форме (без перехода во вторую и третью) слабо пригодно для внедрения в какую-либо информационную систему, так как она не устраняет проблему избыточности данных и может создавать так называемые аномалии обновления.

Аномалии обновления — это общее название группы проблем, которые возникают при проведении операций обновления данных (INSERT, UPDATE, DELETE) в отношениях, имеющих избыточные данные. Чаще всего такие проблемы выливаются в непредсказуемое поведение системы после обновления некоторых данных, так как чтобы обновить один аспект описываемой сущности необходимо провести операцию обновления в нескольких несвязанных друг с другом местах базы данных.

#### 2.3.2 Вторая нормальная форма (2НФ)

Вторая нормальная форма является развитием первой нормальной формы, где добавляется требование наличия у элементов кортежа зависимости от всего первичного ключа целиком. Вот ее полное определение:

Переменная отношения находится во второй нормальной форме тогда и только тогда, когда она находится в первой нормальной форме и каждый неключевой атрибут неприводимо зависит от ее первичного ключа [2].

Нормализация до второй формы призвана упразднить ситуации, когда отношение имеет составной первичный ключ, а значение некоторого неключевого поля зависит от одной части первичного ключа, но не зависит от другой. Сама процедура нормализации состоит в замене текущей ненормализованной переменной отношений на несколько эквивалентных проекций, дающих в сумме ту же переменную отношений (необходимо помнить, что нормализация должна проходить без потери данных).

То есть в ситуации, когда мы имеем переменную отношений A с полями B, C, D, E, где B и C дают составной первичный ключ, а D имеет функциональную зависимость от B, но не имеет от C — правильно будет разложить A на две проекции: A1 и A2, где A1 содержит в себе поля B и D, а A2 — B, C, E. При этом поле B в проекции A2 должно быть не только частью первичного ключа, но и внешним ключом, ссылающимся на таблицу A1.

Данный способ нормализации позволяет избежать прямой зависимости поля D от поля C, при этом исходную переменную отношений можно легко восстановить, то есть данные не потеряны.

#### 2.3.3 Третья нормальная форма (3НФ)

Третья нормальная форма в классическом виде является развитием второй нормальной формы, однако добавляет требование ликвидации транзитивных зависимостей. Ей можно дать такое определение:

Переменная отношения находится в третьей нормальной форме тогда и только тогда, когда она находится во второй нормальной форме и ни один неключевой атрибут не является транзитивно зависимым от ее первичного ключа [2].

Нормализация до третьей нормальной формы призвана упразднить ситуации, когда одно неключевое поле зависит от другого неключевого поля. Метод нормализации напоминает метод нормализации до второй нормальной формы. Для этого так же потребуется разбиение переменной отношения на несколько проекций. Так допустим существует переменная отношения A с полями B, C и D, где поле B является первичным ключом. Кроме этого существует функциональная зависимость поля D от поля C. Одним из возможных вариантов разрешения этой проблемы является разбиение переменной отношения A на проекции A1 и A2, где проекция A1 содержит поля C и D, при этом поле C является первичным ключом; а проекция A2 содержит поля B и C, где B является первичным ключом, а C внешней ключом, ссылающимся на проекцию A1.

#### 2.3.4 Нормальная форма Бойса-Кодда (НФБК)

Третья нормальная форма хорошо работает для большинства баз данных, однако при стечении довольно специфических обстоятельств может оказаться неудовлетворительной. В частности, оно неадекватно при выполнении следующих условий, касающихся определенной переменной отношения:

* переменная отношения имеет два (или больше) потенциальных ключа;
* эти потенциальные ключи являются составными;
* два или больше потенциальных ключей перекрываются (т.е. имеют по крайней мере один общий атрибут) [2].

В этом случае необходимо использование более строгих условий нормализации, то есть нормальной формы Бойса-Кодда. Для того чтобы дать определение НФБК необходимо сначала определить понятия детерминанта и тривиальной функциональной зависимости.

Пусть R является переменной отношения, а X и Y — произвольными подмножествами множества атрибутов переменной отношения R. Y функционально зависимо от X тогда и только тогда, когда для любого допустимого значения переменной отношения R, если два кортежа переменной отношения R совпадают по значению X, они также совпадают и по значению Y. Подмножество X называют детерминантом, а Y — зависимой частью.

Тривиальной функциональной зависимостью называют такою функциональную зависимость, что ее левая часть (детерминант) является надмножеством правой части (зависимой). Тогда нормальную форму Бойса-Кодда можно определить следующим образом:

Переменная отношения находится в нормальной форме Бойса-Кодда тогда и только тогда, когда каждая ее нетривиальная и неприводимая слева функциональная зависимость имеет в качестве своего детерминанта некоторый потенциальный ключ [2].

Заметим, что нормальная форма Бойса-Кодда не имеет в своей формулировке явных отсылок к первой и второй нормальным формам, а также упоминания концепции транзитивных зависимостей. Это делает ее несколько проще для понимания.

#### 2.3.5 Четвертая нормальная форма (4НФ)

Четвертая, пятая и последующие нормальные формы не так сильно распространены, как первые четыре. Это связано с тем, что для большинства информационных систем дальнейшая нормализация оказывается избыточной и не несет ощутимых преимуществ. Однако для полноты картины следует рассмотреть все существующие нормальные формы.

Для определения четвертой нормальной формы необходимо сначала дать определение понятию многозначной зависимости. Пусть R - переменная отношения, а А, B и C являются произвольными подмножествами множества атрибутов переменной отношения R. Тогда подмножество B многозначно зависит от подмножества А. Многозначную зависимость можно считать обобщением понятия функциональной зависимости в том смысле, что каждая функциональная зависимость является также многозначной зависимостью, но обратное утверждение неверно (поскольку существуют многозначные зависимости, которые не являются функциональными). Тогда четвертой нормальной форме можно дать следующее определение:

Переменная отношения R находится в четвертой нормальной форме (4НФ) тогда и только тогда, когда в случае существования таких подмножеств А и В атрибутов этой переменной отношения R, для которых выполняется нетривиальная многозначная зависимость B от A, все атрибуты переменной отношения R также функционально зависят от атрибута А [2].

Говоря более неформально можно сказать, что переменная отношения находится в четвертой нормальной форме тогда и только тогда, когда она находится в нормальной форме Бойса-Кодда и все ее многозначные зависимости фактически являются функциональными зависимостями от ее ключей.

#### 2.3.6 Проекционно-соединительная нормальная форма (5НФ)

Как и в случае с четвертой нормальной формой, для определения пятой нормальной формы необходимо дополнительное понятие зависимости соединения.

Пусть R — переменная отношения, а А, B, ..., Z — произвольные подмножества множества ее атрибутов. Переменная отношения R удовлетворяет следующей зависимости соединения тогда и только тогда, когда любое допустимое значение переменной отношения R эквивалентно соединению ее проекций по подмножествам А, B, ..., Z множества атрибутов. В таком случае определить проекционно-соединительную нормальную форму можно следующим образом.

Переменная отношения R находится в пятой нормально форме (5НФ), которую иногда иначе называют проекционно-соединительной нормальной формой (ПСНФ), тогда и только тогда, когда каждая нетривиальная зависимость соединения в переменной отношения R определяется потенциальным ключом (ключами) R, если соблюдаются приведенные ниже условия:

1. Зависимость соединения \*{ А, B, ..., Z } в переменной отношения R является тривиальной тогда и только тогда, когда по крайней мере одно из подмножеств А, B, ..., Z множества атрибутов является множеством всех атрибутов R.
2. Зависимость соединения \*{ А, B, ..., Z } в переменной отношения R определяется потенциальным ключом (ключами) R тогда и только тогда, когда каждое из подмножеств А, B, ..., Z множества атрибутов является суперключом для R [1].

Кроме этого необходимо помнить, что находясь в пятой нормальной форме переменная отношения находится и в четвертой нормальной форме так как многозначная зависимость является частным случаем зависимости соединения [2].

Фактически считается, что пятая нормальная форма является последней, так как на этом этапе нормализации уже гарантируется, что переменная отношений свободна от аномалий, которые могут быть исключены посредством ее разбиения на проекции. Однако это не означает, что не существует других нормальных форм.

#### 2.3.7 Доменно-ключевая нормальная форма (ДКНФ)

Доменно-ключевая нормальная форма является довольно экзотическим представителем нормальных форм, так как в ее формулировке не используются понятия функциональных зависимостей, многозначных зависимостей или зависимостей соединения. Вместо этого используются понятия ограничения доменов и ограничения ключей.

Ограничениями доменов называют ограничение, которое предписывает использование для определенного атрибута значений только из некоторого заданного домена.

Ограничениями ключей представляет собой предписание использовать некоторый атрибут или множество атрибутов в качестве потенциального ключа. Тогда формулировка доменно-ключевой нормальной формы будет звучать следующим образом.

Утверждается, что переменная отношения R находится в ДКНФ тогда и только тогда, когда каждое наложенное на нее ограничение является логическим следствием ограничений доменов и ограничений ключей, наложенных на данную переменную отношения [2].

Стоит также помнить, что если переменная отношения находится в доменно-ключевой нормальной форме, то она находится и в пятой нормальной форме (соответственно и во всех остальных). Однако не всегда можно привести переменную отношения к доменно-ключевому виду и даже не всегда можно сказать, можно ли выполнить такое приведение.

#### 2.3.8 Шестая нормальная форма (6НФ)

Шестая нормальная форма существует как обобщение для пятой и полезна в целом только когда речь идет о хронологических базах данных.

Хронологическая база данных может быть неформально определена как база, которая содержит исторические данные наряду с текущими данными или вместо них. Обычные, или нехронологические, базы данных содержат только текущие данные; актуальность таких баз поддерживается путем обновления данных сразу же после того, как представленные в них высказывания становятся ложными. В отличие от них, хронологические базы данных обновляются очень редко (а могут вообще не обновляться), если не считать выполнения операций INSERT, которые применяются для их первоначального заполнения [2].

Хронологические базы данных хранят информацию о текущих состояниях, прошлых состояниях и, вероятно, будущих состояниях. Так например база данных может хранить информацию о правах пользователя на доступ к какому либо контенту в формате: пользователь A получал право доступа с 30 января по 7 февраля, с 14 марта по 15 мая и получит 6 сентября, а потеряет 12 октября.

Особенностью шестой нормальной формы является то, что декомпозиция проводится до тех пор, пока это в принципе возможно без потерь. Для нехронологических баз данных подобный подход не имеет особого смысла и скорее даже вреден, поэтому для них пятая нормальная форма является конечной. Кроме этого, для нормализации хронологических баз данных используется специальный метод вертикальной декомпозиции, который так же слабо применим к нехронологическим базам данных. Итак, получим определение шестой нормальной формы.

Переменная отношения находится в шестой нормальной форме тогда и только тогда, когда она удовлетворяет всем нетривиальным зависимостям соединения. Из определения следует, что переменная находится в 6НФ тогда и только тогда, когда она неприводима, то есть не может быть подвергнута дальнейшей декомпозиции без потерь. Каждая переменная отношения, которая находится в 6НФ, также находится и в 5НФ [2].

#### 2.4 Язык SQL

Сегодня язык SQL является стандартным средством для работы с реляционными базами данных. Изначально он разрабатывался Язык SQL первоначально разрабатывался конкретно как подъязык данных. Однако после включения в стандарт в конце 1996 года такого средства, как постоянные хранимые модули SQL (SQL Persistent Stored Modules — SQL/PSM, или сокращенно PSM), стандарт SQL стал полностью поддерживать все вычислительные конструкции.

В языке SQL вместо привычных уже терминов отношение и переменная отношения используется термин таблица, а вместо терминов кортеж и атрибут — строка и столбец [2].

В данной главе будут рассмотрены основные составляющие элементы языка SQL: типы данных, различные типы операций над данными и виды представления и хранения SQL кода.

Начнем с определения основных понятий синтаксиса языка SQL. Элементарными единицами синтаксиса SQL являются ключевые слова и идентификаторы. Ключевым словом в SQL называют такое слово значение которого в рамках языка строго определено, использование таких слов в неверном контексте будет считаться ошибкой. Идентификаторами же называются такие слова, которые используются для именования объектов базы данных, с которыми ведется работа. Такие слова уже могут быть произвольными, однако существуют некоторые общепринятые стандарты по их написанию, которым желательно следовать.

По одиночке ключевые слова и идентификаторы мало на что способны, для полноценного функционирования языка необходимы минимальные законченные языковые конструкции, их называют предложениями. Множество предложений, работающих вместе называют командой SQL, любая команда SQL должна содержать как минимум одно предложение [4]. Отделение различных команд SQL друг от друга происходит с помощью точки с запятой. Сами команды могут иметь довольно свободное начертание до тех пор пока ключевые слова и идентификаторы не разбиваются по слогам.

#### 2.5 Типы данных в языке SQL

Как и любой другой язык, SQL умеет различать многие типы данных: от строковых и числовых, и до интервальных. Перечень типов данных, вообще говоря, определяется базой данных, которая используется в информационной системе, однако можно выделить несколько общих видов типов данных, которые в том или ином виде встречаются везде.

#### 2.5.1 Строковые типы данных

Строковые типы данных служат в SQL для представления текста. Они являются упорядоченным количеством некоторого (возможно, нулевого) количества символов [4].

Строковые типы данных легко конвертируются в другие типы данных, если используется соответствующая форма записи строки, а также парсятся из других типов данных, причем в этом случае требования к форме значительно более слабые.

Строковые типы данных обладают следующими характеристиками:

* длина объектов может быть как фиксированной, так и переменной;
* учитывает регистр (например, при сортировке 'A' стоит перед 'a');
* в командах SQL любой строковый литерал должен быть заключен в одинарные кавычки.

Самыми известными представителями строковых типов являются: text, varchar, char.

#### 2.5.2 Битовые типы данных

Если строковые типы данных нужны для хранения именно текста, то в битовые типы данных могут хранить практически любые объекты, так как в битовом виде можно представить любую информацию. Типы данных этого вида являются упорядоченной последовательностью, состоящей из какого-то числа, возможно из 0, битов [4].

Почти абсолютная гибкость этих типов данных уравновешивается тем, что людям их прочитать невозможно. Эта проблема не позволяет применять данную группу типов данных повсеместно и даже делает ее довольно специфичным решением или крайней мерой. Так например, если в обычной реляционной базе данных необходимо хранить не очень большой файл, то для этого подойдет тип BLOB (Binary Large OBject). Этот тип имеет несколько модификаций в зависимости от базы данных, в которой он используется, но суть его — хранение большого большого бинарного массива, не меняется.

В командах SQL любую строку битов (как и обычный текст) следует заключать в одинарные кавычки.

Наиболее распространенными битовыми типами данных являются blob (tinyblob, mediumblob, largeblob), bit, varbit и raw.

#### 2.5.3 Точные численные типы данных

В языке SQL имеются несколько групп типов данных для представления чисел. Первая группа — типы данных для представления целых (точных) чисел.

Точные численные типы данных представляют некоторое число, которое может быть положительным, отрицательным или нулем, кроме того, точные численные типы могут использоваться для отображения как целых, так и рациональных чисел.

Основной особенностью точных численных типов является, то что они могут иметь произвольные, но при этом строго фиксированные масштаб и точность [4].

Точностью называется число значащих цифр в записи числа (то есть общее количество цифр в десятичной записи числа без учета десятичной точки. Масштабом называется число цифр справа от десятичной точки (очевидно, что для получения целого числа необходимо и достаточно установить масштаб равным 0, а также то, что масштаб не может быть больше точности).

Наиболее известными представителями точных численных типов являются: byte, numeric, integer, decimal и другие.

#### 2.5.4 Действительные числовые типы данных

Действительные числовые типы нужны для представления некоторых аппроксимаций точных числовых значений. Они так же могут представлять собой положительные и отрицательные числа, либо ноль, но при этом представляют собой рациональное приближение некого действительного числа с плавающей точкой. Такие числа обычно применяют, чтобы формализовать или очень маленькие или, наоборот, очень большие количества чего-либо, или какие-нибудь научные вычисления [4].

Выражаются они как правило в экспоненциальном виде с использованием мантиссы и порядка. Сама запись представляет собой некоторое рациональные число, умноженное на десять в некоторой степени.

Действительные числовые типы данных, в отличии от точных, не имеют фиксированного масштаба, хотя и определяются со строго выбранной точностью.

К действительным числовым типам данных относятся: float, real, double, single.

#### 2.5.5 Календарные типы данных

Кроме стандартных типов данных для отображения чисел, текста или двоичного кода, для любой базы данных фактически необходимо иметь типы данных для отображения даты и времени. Без таких типов становится невозможным удобная и точная расстановка временных маркеров в записях таблиц, а также оперирование данными о датах в функциях и процедурах.

Для решения этих задач существуют календарные типы данных, которые хранят данные о дате и времени в стандартизированной форме.

Время в таких типах данных рассчитывается по соотношению к UTC, при этом, согласно стандарту SQL-92, каждому сеансу обращения к базе данных выставляется свой сдвиг по UTC, который соответствует текущему часовому поясу пользователя и учитывается функциями получения текущей даты и времени, а так же ее адаптации для часовых поясов других регионов. Расчет времени происходит в 24-часовом формате с разделением значимых частей записи символом двоеточия.

Также в календарных типах данных поддерживаются только даты Григорианского календаря, коммерческие, на которых сегодня строится абсолютное большинство современных промышленных систем не поддерживают работу с календарями, составленными по другим правилам. Формат записи даты может сильно отличаться и требует проверки документации каждой базы данных отдельно, однако зачастую дату записывают в формате YYYY-MM-DD.

Календарные типы данных выражены в типах datetime, timestamp и timestamp with timezone.

#### 2.5.6 Интервальные типы данных

Интервальные типы данных являются производными от календарных типов и показывают расстояние между двумя датами. Интервальные типы данных хранят количественную меру промежутка времени между двумя значениями даты или/и времени суток так, что если вы вычтете из одного значения другое, то получите этот самый интервал.

Эти типы содержат те же поля, что и datetime и имеет те же разделители, однако значения интервальных типов данных имеет направление, направление интервала может идти вперед (со знаком плюс) и назад (со знаком минус). Кроме этого интервальные типы отличаются по форме записи и уровням измерения времени.

Квалификатор интервальных типов данных может быть однополевым и пногополевым. Однополевой квалификатор нужен для хранения интервала в однородных единицах измерения (1 час; 2 минуты; 5 секунд). Многополевые квалификаторы допускают использование смешанных единиц измерения времени (1 час, 2 минуты, 5 секунд).

Уровни измерения времени бывают Year-month и Day-time. Year-month интервалы нужны для выражения крупных промежутков времени в месяцах и годах. Day-time интервалы для более мелких отрезков, состоящих из дней, часов и минут.

#### 2.6 Хранимый код в языке SQL и его диалектах

Стандартный язык SQL хорошо подходит для не слишком сложных задач по работе с информацией, однако современные большие базы данных требуют гораздо большего уровня автоматизации внутренних процессов и гораздо более сложных алгоритмов для работы с информацией. Для этих целей крупные поставщики промышленных баз данных, таких как Oracle, PostgreSQL и MySQL, имеют собственные диалекты языка SQL, которые позволяют пользователю писать хранимые процедуры, функции, триггеры и многое другое. Рассмотрим работу этих механизмов на примере базы PostgreSQL и ее языка PL/pgSQL.

#### 2.6.1 Хранимые процедуры и функции в языке PL/pgSQl

В начале необходимо четко отделить понятие хранимой процедуры от понятия функции, так как часто их используют как взаимозаменяемые, что является ошибкой.

Функция — это лишь некоторая часть SQL команды, ей не позволяется инициировать или фиксировать транзакцию. Это сделано для соблюдения транзакционной целостности и непротиворечивости.

Процедуры же наоборот могут управлять транзакциями и даже запускать несколько транзакций последовательно. Но, в отличии от функций, процедуры нельзя вызвать через SELECT, вместо этого следует использовать специальную команду CALL [5].

Кроме процедур и функций код на языке PL/pgSQL может записываться в виде анонимных блоков кода. Анонимные блоки кода, в отличии от обычной SQL команды, имеют структуру, свойственную более привычным языкам программирования и зачастую легче для чтения. Их особенность состоит в том, что они не принимают аргументов, не имеют имени и не хранятся в базе данных. Нужны анонимные блоки кода строго для однократного использования.

Для создания процедур и функций в PL/pgSQl используются команды CREATE PROCEDURE и CREATE FUNCTION соответственно. Синтаксически они очень похожи и в усредненном виде выглядят следующим образом.

CREATE [ OR REPLACE ] PROCEDURE

имя ( [ [ режим\_аргумента ] [ имя\_аргумента ] тип\_аргумента

[ { DEFAULT | = } выражение\_по\_умолчанию ]

[, ...] ] )

{ LANGUAGE имя\_языка

| TRANSFORM { FOR TYPE имя\_типа } [, ... ]

| [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER

| SET параметр\_конфигурации { TO значение | = значение | FROM CURRENT }

| AS 'определение'

| AS 'объектный\_файл', 'объектный\_символ'

} ...

Хранимые процедуры являются неотъемлемой частью языка PL/pgSQL и значительно повышают эффективность разработки ПО [5].

#### 2.6.2 Триггеры в языке PL/pgSQl

Одной из наиболее важных задач, которая решается серверным кодом является реакция на изменение в базе данных. Это означает срабатывание некоторого алгоритма при выполнении SQL команд INSERT, UPDATE, DELETE или TRUNCATE. Вызываемая триггером функция может модифицировать изменившиеся данные или просто выполнить необходимую операцию.

В PostgreSQL для одной таблицы можно определить сколь угодно много триггеров. Порядок срабатывания определен в PostgreSQL 7.3: триггеры срабатывают в алфавитном порядке, при этом сначала работают триггеры определенные через команду BEFORE (они срабатывают перед проведением операции над данными), после запускаются триггеры определенные через AFTER (данные триггры срабатывают, соответственно, после изменения данных) [5].

Общая форма определения триггера в PL/pgSQL выглядит следующим образом.

CREATE [ CONSTRAINT ] TRIGGER имя { BEFORE | AFTER | INSTEAD OF } { событие [ OR ... ] }

ON имя\_таблицы

[ FROM ссылающаяся\_таблица ]

[ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]

[ REFERENCING { { OLD | NEW } TABLE [ AS ] имя\_переходного\_отношения } [... ] ]

[ FOR [ EACH ] { ROW | STATEMENT } ]

[ WHEN ( условие ) ]

EXECUTE PROCEDURE имя\_функции ( аргументы )

Триггеры — это хороший способ проверить данные и завершить операцию с ошибкой, если нарушены какие-то пользовательские ограничения.

#### 2.7 Индексы в языке SQL

При работе с большими базами данных зачастую возникает проблема снижения скорости ее работы из-за большого количества однокипных запросов к одним и тем же полям одних и тех же таблиц. Для ускорения этого поиска и создается индекс.

Индекс организован таким образом: на основе данных, содержащихся в конкретной строке таблицы, формируется значение элемента (записи) индекса, соответствующего этой строке. Для поддержания соответствия между элементом индекса и строкой таблицы в каждый элемент помещается указатель на строку [6].

Индекс является упорядоченной структурой. Элементы (записи) в нем хранятся в отсортированном виде, что значительно ускоряет поиск данных в индексе. После отыскания в нем требуемой записи СУБД переходит к соответствующей строке таблицы по прямой ссылке. Записи индекса могут формироваться на основе значений одного или нескольких полей соответствующих строк таблицы. Значения этих полей могут комбинироваться и преобразовываться различными способами.

При выполнении поиска конкретных строк в таблице специальная подсистема СУБД, называемая планировщиком, проверяет, имеется ли для этой таблицы индекс, созданный на основе тех же столбцов, что указаны, например, в условии предложения WHERE. Если такой индекс существует, то планировщик оценивает целесообразность его использования в данном конкретном случае. Если его использование целесообразно, то сначала выполняется поиск необходимых значений в индексе, а затем, если такие значения в нем найдены, производится обращение к таблице с использованием указателей, которые хранятся в записях индекса. Таким образом, полный перебор строк в таблице может быть заменен поиском в упорядоченном индексе и переходом к строке таблицы по прямому указателю (ссылке) [7].

Индексы создаются с помощью следующей команды.

CREATE INDEX имя-индекса ON имя-таблицы ( имя-столбца, ...);

#### 2.8 Нереляционные базы данных

Нереляционные базы данных — это крайне широкое понятие, включающее в себя огромное количество самых разнообразных систем хранения данных. В данной главе будут рассмотрены документо-ориентированные и графовые базы данных.

#### 2.9 Документо-ориентированные базы данных

В главе посвященной реляционным базам данных говорилось, что эти системы строятся на реляционной модели данных, которая имплементируется в программный продукт путем создания переменных отношений и картежей. Проблемой этого подхода является то, что картеж довольно ограничен в области хранимых типов данных. Сложные структуры данных вроде списков или словарей могут быть помещены в картеж только после сведения их к более простым типам, что приводит к проблеме потери соответствия [8].

Решением этой проблемы стала разработка баз данных, основанных на документо-ориентированной модели. Основной концепцией в документных базах данных является документ. База данных хранит и извлекает документы в форматах ХМL, JSON, BSON и т.д.. Эти документы представляют собой самоописываемые иерархические древовидные структуры данных, которые могут состоять из ассоциативных массивов, коллекций и скалярных значений.

По-другому документы в NoSQL базах данных называются агрегатами. Данный термин пришел в документо-ориентированное проектирование из объектно-ориентированного проектирование и означает сложную запись, которая может содержать списки и другие структуры записей. Документы или агрегаты составляют ядро NoSQL систем и являются для них тем же, чем являются картежи для реляционных баз данных.

Документы хранятся примерно одинаково, но не обязательно должны быть одинаковыми. Документные базы данных хранят документы в качестве значений в хранилищах типа "ключ-значение"; документные базы данных можно интерпретировать ках хранилища типа "ключ-значение", в которых значение допускает проверку [8].

В документо-ориентированных базах данных отсутствует строгая предопределенность схемы хранения данных. Это несет некоторые преимущества для разработчиков. Во-первых, при таком подходе к проектированию структуру данных будет определять программный код разрабатываемой системы, а не база данных. Во-вторых, документо-ориентированные базы данных позволяют представлять данные с переменным набором свойств [9].

#### 2.10 MongoDB

Сегодня самой популярной документо-ориентированной базой данных на рынке является MongoDB.

MongoDB реализует в себе все необходимые элементы документо-ориентированной базы данных. Основной единицей хранения данных в MongoDB является документ. Все документы, как это и требует основная модель данных, могут иметь абсолютно разную структуру. Документы в MongoDB объединяются в коллекции, и хотя они все так же могут иметь различную структуру, как правило в одной коллекции хранятся документы схожей структуры.

Нереляционные базы данных, в том числе и MongoDB, имеют некоторые ограничения, связанные с выполнением произвольных запросов. В реляционных базах данных за счет строго определенной схемы хранения данных таких проблем не возникает никогда, база выполнит любой правильно составленный SQL запрос. В нереляционных базах данных механизмы произвольных запросов либо отсутствуют вовсе, либо (как в случае с MongoDB) могут быть реализованы только в рамках одного документа. Сам же запрос к базе данных скорее похож на обращение к GraphQL серверу. Выглядят запросы следующим образом [9].

*db.post.find({‘tags’: ‘politics’, vote\_count’: {‘$gt’: 10}});*

Одной из полезных функций MongoDB является возможность индексирования для повышения скорости выполнения запросов. MongoDB поддерживает несколько проверенных временем структур индексов, в том числе классическое B-дерево, а также двумерные и сферические пространственные индексы.

Но стоит учитывать, что из-за отсутствия строгой схемы организации данных создание индекса для большой коллекции займет много времени и системных ресурсов. Это следует учитывать и стараться строить индексы в периоды наименьшей нагрузки, в фоновом режиме и вручную, не применяя автоматизированные процедуры [10].

#### 2.11 Графовые базы данных

Причина появления графовых баз данных в семействе NoSQL систем кроется в особенностях кластеризации документо-ориентированной модели данных в целом. В какой-то момент разработчики пришли к тому, что при кластеризации они получают простой по структуре связей документ, который при этом получается крайне большим. Графовые базы данных имеют противоположную модель — маленькие объекты, но с очень сложными связями.

В таком контексте этот граф — не диаграмма, а структура данных с узлами, соединенными ребрами [8].

Как только построение графовой сети закончено, пользователь может отправлять этой сети запросы на обработку данных. Здесь проявляется главное отличие графовых баз данных, во-первых, от реляционных, во-вторых, от иных NoSQL баз данных — производительность графовой базы данных не падает по мере увеличения количества связей внутри нее. Это связано с тем, что графовые базы данных переносят большую часть работы по навигации на момент вставки, следовательно стоимость перехода между узлами становится крайне низкой.

В графовых базах данных также существуют механизмы индексирования. Выполнены они через присваивание тегов некоторым узлам. Индексы корректируют порядок обхода графа, делая операции сбора нужной информации из сети более оптимальным.

Акцент на связях резко отличает графовые базы данных от любых иных. Это имеет свои преимущества (описаны выше) и свои недостатки. К недостаткам можно отнести крайне тяжелый процесс кластеризации графа. Кластеризация делает процесс обхода графа гораздо более затратным и требует охвата нескольких ребер и узлов процессами вставки для обеспечения согласованности данных [8].

#### 2.12 Neo4j

Сродни MongoDB, которая является самой популярной документо-ориентированной базой данных, Neo4j является самой популярной графовой базой данных.

В дистрибутив Neo4j входят инструменты для ускорения поиска (Lucene) и простые в использовании (хотя, быть может, не вполне привычные) языковые расширения, в частности Gremlin и REST-интерфейс. Но Neo4j не только проста в использовании, но и работает быстро. В отличие от операций соединения в реляционных базах данных или операций map-reduce в других базах, обход графа требует постоянного времени. Связанные данные находятся всего в одном шаге – не нужно производить массивное соединение с последующей фильтрацией результатов, как в большинстве рассмотренных выше СУБД. Каким бы большим ни был граф, переход из узла A в узел B требует всего одного шага, если между этими узлами имеется связь. Наконец, в издание Enterprise включены средства для создания высокодоступных сайтов с большим количеством запросов на чтение – путем создания HA-кластера Neo4j.

В работе Neo4j есть несколько особенностей, про которые следует знать перед началом работы. Ребра в Neo4j не могут входить в ту же вершину, из которой исходят8 . Нам также кажется, что выбранная терминология (узел вместо вершины и связь вместо ребра) только затрудняет взаимопонимание. HA-кластер прекрасно справляется с репликацией, но реплицировать разрешено только граф целиком. Невозможно сегментировать граф, что налагает ограничение на его размер [10].

#### 2.13 Язык Cypher

Neo4j поддерживает множество различных языков обращения к графу, однако стандартным считается язык Cypher. Cypher предоставляет пользователю (или приложению, действующему от имени пользователя) возможность задавать шаблон для поиска данных.

Шаблоны графов, представленные ASCII-графикой, являются основой Cypher. Запросы на Cypher прикрепляют одну или несколько частей шаблона к определенным местам графа с помощью предикатов, а затем перемещают незафиксированные части, пытаясь найти соответствие.

Как и большинство языков запросов, Cypher состоит из фраз. Прос тейшие запросы состоят из фразы MATCH и следующей за ней фразы RETURN (ниже в этой главе будет приведено описание и других фраз, используемых в запросах на Cypher). Следующий пример запроса на Cypher использует эти фразы для поиска прямых друзей пользователя Jim [11].

MATCH (a:Person {name:'Jim'})-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-[:KNOWS]->(c) RETURN b, c.

#### 2.14 Выбор базы данных

После анализа преимуществ и недостатков изученных баз данных было принято решение взять реляционную базу данных, как основную базу данных проекта. Одним из стандартных решений для такой задачи является PostgreSQL.

Соответственно, языком написания функций, процедур и триггеров будет язык PL/pgSQL. Этот язык является диалектом языка SQL, который используется в PostgreSQL.

#### 3 ПРОЕКТИРОВАНИЕ БАЗЫ ДАННЫХ

Перед созданием физической базы данных для проекта необходимо изучить предметную область и выделить необходимые сущности, которые будут описаны отношениями в базе данных.

Для более наглядной демонстрации и систематизации архитектуры будут составлены схемы.

#### 3.1 Разработка структуры базы данных

Для функционирования разрабатываемого проекта с заданным функционалом потребуется следующий набор отношений (без учеты связывающих отношений).

1. advertisement — информация о рекламных кампаниях и рекламных блоках:
   * ad\_name — название кампании;
   * start\_date — начало кампании;
   * end\_date — конец кампании;
   * cost — стоимость компании.
2. album — сборник треков одного исполнителя или группы:
   * title — название альбома;
   * artist\_id — исполнитель.
3. artist — информация об исполнителе или группе, включая имя, жанр, биографию и так далее:
   * name — имя исполнителя или название группы;
   * genre\_id — основной жанр;
   * biography — описание исполнителя.
4. chart — списки лучших треков, альбомов или исполнителей:
   * name — название чарта;
   * creation\_date — время создания.
5. content\_rights — информация о правах на музыкальный контент:
   * track\_id — ссылка на объект контента;
   * rights\_holder — правообладатель;
   * rights\_type — тип прав;
   * start\_date — дата вступления прав в силу;
   * end\_date — дата окончания правообладания.
6. events — информация о музыкальных событиях, таких как концерты и фестивали:
   * name — название события;
   * location — место проведения события:
   * event\_date — дата проведения события.
7. genre — музыкальный жанр:
   * title — название жанра.
8. gift\_cards\_and\_codes — специальные коды и акции:
   * code — код;

* value — значение кода;
  + expiration\_date — дата истечения кода.

1. listening\_history — история прослушивания песен пользователя:
   * user\_id — ссылка на пользователя;
   * track\_id — ссылка на композицию;
   * listening\_date — дата прослушивания.
2. listening\_statistics — статистические данные о прослушивании треков, альбомов и исполнителей:
   * user\_id — ссылка на пользователя;
   * track\_id — ссылка на композицию;
   * listening\_count — количество прослушиваний.
3. notifications — информация о уведомлениях, отправленных пользователю:
   * user\_id — ссылка на пользователя;
   * notification\_text — описание уведомления;
   * notification\_date — дата отправки уведомления.
4. payment — информация о платежах пользователя:
   * user\_id — ссылка на пользователя;
   * amount — сумма оплаты;
   * payment\_date — дата проведения транзакции.
5. playlist — список треков, созданный пользователем или сервисом:
   * user\_id — ссылка на пользователя;
   * title — название плейлиста.
6. podcast — информация о подкастах, доступных на сервисе:
   * title — название подкаста;
   * description — описание подкаста;
   * release\_date — дата размещения.
7. radio\_station — информация о радиостанциях, доступных на сервисе:
   * name — название радиостанции;
   * genre\_id — основной жанр радиостанции.
8. rating — оценка трека или альбома, данная пользователем:
   * user\_id — ссылка на пользователя;
   * track\_id — ссылка на композицию;
   * score — оценка.
9. recommendation — список рекомендованных песен, альбомов или исполнителей для пользователя:
   * user\_id — ссылка на пользователя;
   * track\_id — ссылка на композицию;
   * recommendation\_date — дата выставления рекомендации.
10. review — отзыв пользователя на трек, альбом или исполнителя:
    * user\_id — ссылка на пользователя;
    * track\_id — ссылка на композицию;
    * text — текст отзыва.
11. securityinfo — информация о безопасности аккаунта пользователя, включая данные для входа, историю входа и другое:
    * user\_id — ссылка на пользователя;
    * login\_data — дата входа;
    * login\_history — история входа.
12. social\_media — информация о социальных сетях, связанных с аккаунтом пользователя:
    * user\_id — ссылка на пользователя;
    * platform\_name — название платформы;
    * profile\_link — ссылка на профиль.
13. subscription — информация о подписке пользователя, включая тип подписки, стоимость, дату начала и окончания:
    * user\_id — ссылка на пользователя;
    * type — тип подписки;
    * cost — стоимость подписки;
    * start\_date — начало действия подписки;
    * end\_date — конец действия подписки.
14. support — информация о запросах в службу поддержки от пользователя:
    * user\_id — ссылка на пользователя;
    * request\_text — текст запроса;
    * request\_date — дата запроса;
    * response\_text — текст ответа;
    * response\_date — дата ответа.
15. track — информация о музыкальном произведении, включая название, исполнителя, альбом, жанр:
    * title — название произведения;
    * artist\_id — ссылка на артиста;
    * album\_id — ссылка на альбом;
    * genre\_id — ссылка на доминирующий жанр.
16. user\_settings — настройки пользователя, включая предпочтения в музыке, настройки уведомлений:
    * user\_id — ссылка на пользователя;
    * music\_preferences — музыкальные настройки;
    * notification\_settings — настройки уведомлений.
17. users — информация о пользователе, включая имя, адрес электронной почты, дату регистрации:
    * name — имя пользователя;
    * email — почта пользователя;
    * registration\_date — дата регистрации;
    * пароль — пароль для входа в учетную запись.

#### 3.2 Создание ERD диаграмм

ERD диаграммы используются для демонстрации связи сущностей внутри системы. Диаграммы делятся на три уровня, соответствующих уровню описываемой абстракции: концептуальный — самый высокий уровень абстракции, необходимо для представления системы целиком; логический — уровень, демонстрирующий связь сущностей внутри баз данных; физический — уровень, описывающий структуру баз данных со всеми техническими подробностями.

В рамках работы над данным проектом будет разработана логическая и физическая ERD диаграммы. Логическая ERD диаграмма представлена на рисунке 3.1.

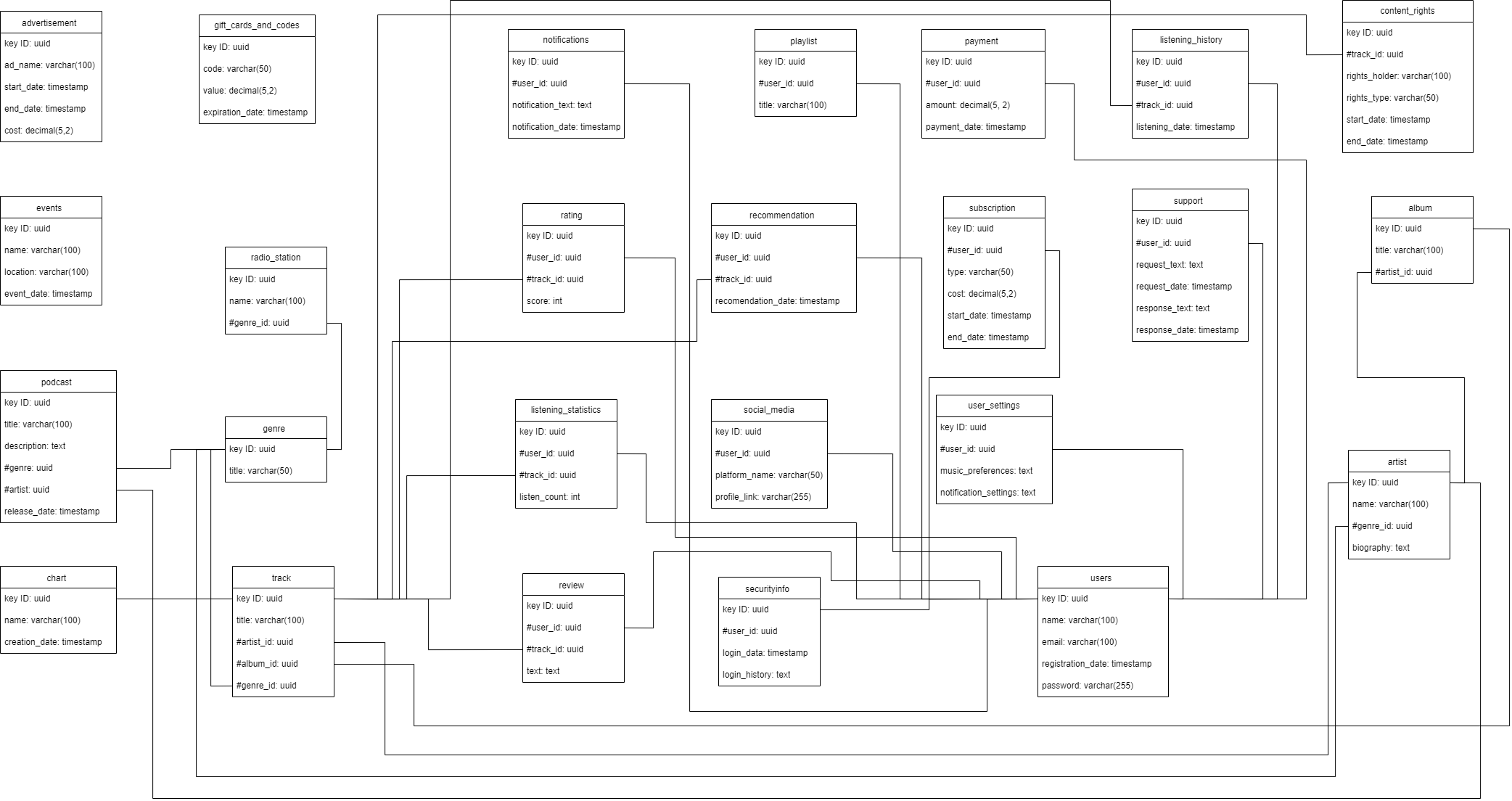


Рисунок 3.1 — Логическая ERD диаграмма

В продолжение проектирование создается физическая ERD схема, которая будет использована непосредственно в процессе создания физической базы данных. Физическая ERD диаграмма представлена на рисунке 3.2.

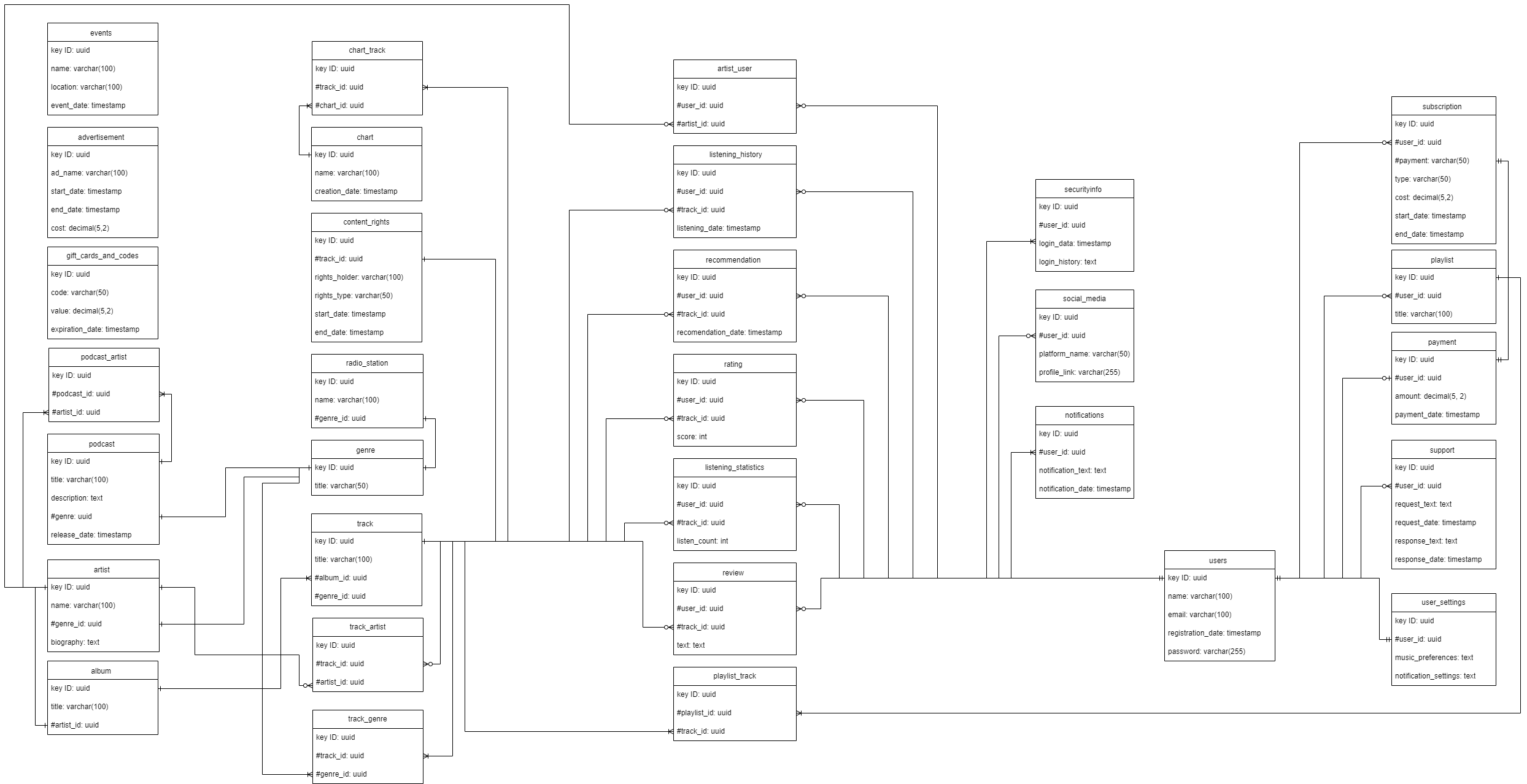


Рисунок 3.2 — Физическая ERD диаграмма

#### 3.3 Создание IDEF1X диаграммы

IDEF1X диаграммы служат для облегчения понимания физических схем баз данных путем добавления семантических обозначений, указывающих на характер взаимодействия информационных объектов в ней. В сущности, она добавляет один уровень абстракции к низкоуровневой схеме базы данных, что позволяет техническим разработчикам и бизнес-аналитикам одинаково продуктивно взаимодействовать с данными схемами. IDEF1X диаграмма базы данных проекта представлена на рисунке 3.3.

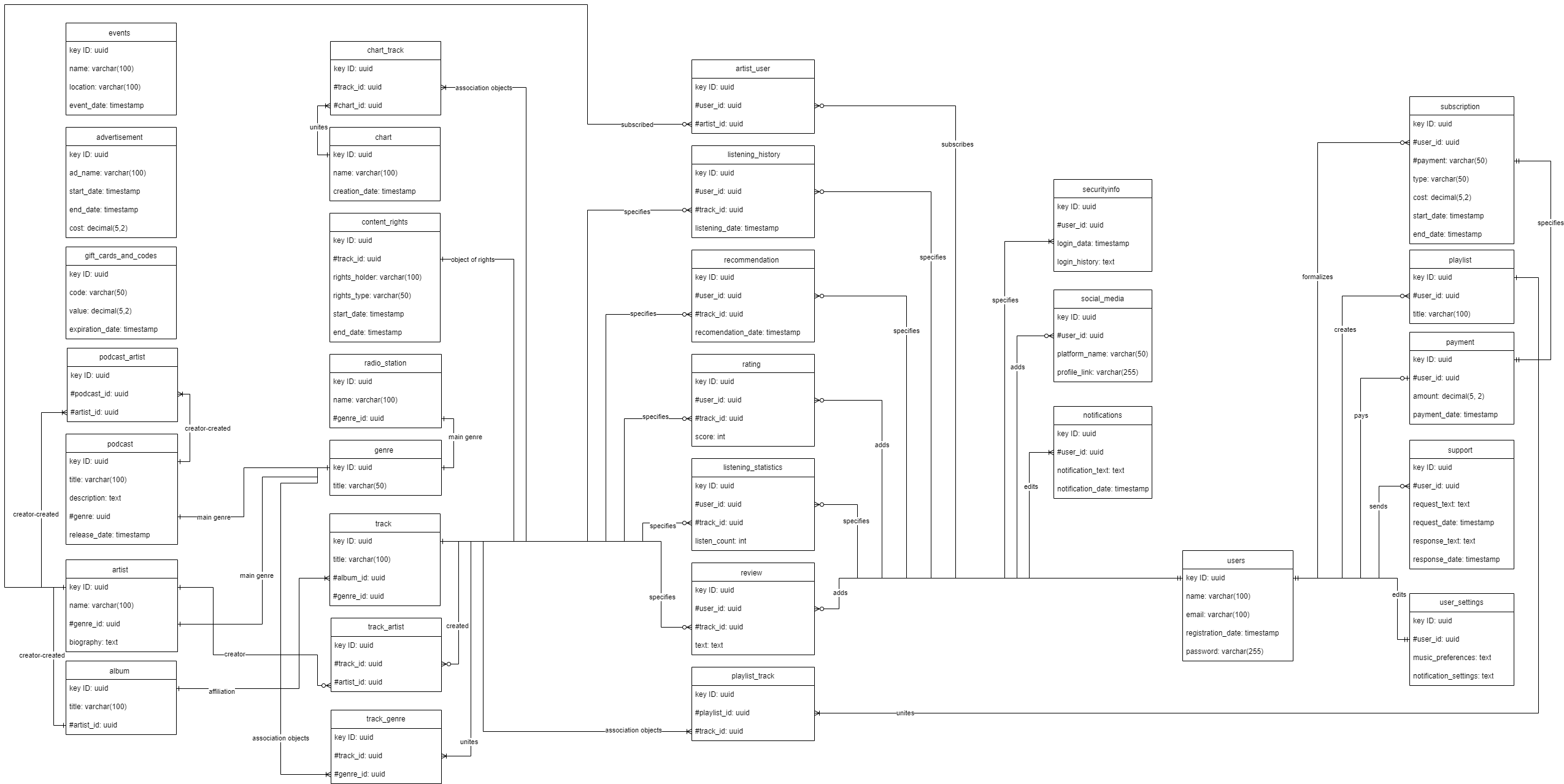


Рисунок 3.3 — IDEF1X диаграмма базы данных проекта

#### 4 РАЗРАБОТКА ФИЗИЧЕСКОЙ БАЗЫ ДАННЫХ

Имея созданные схемы базы данных, выбранную технологию построения базы и рабочий диалект языка SQL можно приступить к созданию базы данных.

Программный SQL-код, реализующий данные сущности представлен в данной главе и приложении к данной пояснительной записке.

#### 4.1 Создание функций

Исходный код функций представлен в приложении к настоящей работе.

**4.1.1** Функция для подсчета песен в плейлисте

Эта процедура возвращает количество песен в заданном плейлисте. На вход подается идентификатор плейлиста, на выходе получается полная выборка из отношения track. Код данной функции:

CREATE OR REPLACE FUNCTION count\_tracks\_in\_playlist(p\_playlist\_id INT) RETURNS INT AS $$

DECLARE

track\_count INT;

BEGIN

SELECT COUNT(\*) INTO track\_count FROM Playlist\_Track WHERE playlist\_id = p\_playlist\_id;

RETURN track\_count;

END;

$$ LANGUAGE plpgsql;

**4.1.2** Функция для получения плейлистов пользователя

Эта процедура возвращает все плейлисты заданного пользователя. На вход подается идентификатор пользователя, на выходе получается полная выборка из отношения playlist. Код данной функции:

CREATE OR REPLACE FUNCTION get\_user\_playlists(p\_user\_id INT) RETURNS TABLE(playlist\_id INT, title VARCHAR(100)) AS $$

BEGIN

RETURN QUERY SELECT id, title FROM Playlist WHERE user\_id = p\_user\_id;

END;

$$ LANGUAGE plpgsql;

**4.1.3** Функция для получения топ-10 песен по жанру

Эта процедура возвращает топ-10 треков заданного жанра по среднему рейтингу. На вход подается идентификатор жанра, на выходе получается полная выборка из отношения track. Код данной функции:

CREATE OR REPLACE FUNCTION get\_top\_tracks\_by\_genre(p\_genre\_id INT) RETURNS TABLE(track\_id INT, title VARCHAR(100), average\_rating DECIMAL(3,2)) AS $$

BEGIN

RETURN QUERY SELECT id, title, average\_rating FROM Track WHERE genre\_id = p\_genre\_id ORDER BY average\_rating DESC LIMIT 10;

END;

$$ LANGUAGE plpgsql;

**4.1.4** Функция для получения всех альбомов исполнителя

Эта процедура возвращает все альбомы заданного исполнителя. На вход подается идентификатор артиста, на выходе получается полная выборка из отношения album. Код данной функции:

CREATE OR REPLACE FUNCTION get\_artist\_albums(p\_artist\_id INT) RETURNS TABLE(album\_id INT, title VARCHAR(100)) AS $$

BEGIN

RETURN QUERY SELECT id, title FROM Album WHERE artist\_id = p\_artist\_id;

END;

$$ LANGUAGE plpgsql;

#### 4.2 Создание триггеров

Исходный код триггеров представлен в приложении к настоящей работе.

**4.2.1** Триггер для проверки дубликатов email

Этот триггер предотвращает создание двух пользователей с одинаковыми адресами электронной почты. Код данного триггера:

CREATE OR REPLACE FUNCTION check\_email\_duplicates() RETURNS TRIGGER AS $$

BEGIN

IF EXISTS (SELECT 1 FROM users WHERE email = NEW.email) THEN

RAISE EXCEPTION 'Пользователь с email % уже существует', NEW.email;

END IF;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER email\_trigger

BEFORE INSERT OR UPDATE ON users

FOR EACH ROW EXECUTE PROCEDURE check\_email\_duplicates();

**4.2.2** Триггер для обновления даты последнего прослушивания

Этот триггер автоматически обновляет дату последнего прослушивания трека при добавлении записи в историю прослушивания. Код данного триггера:

CREATE OR REPLACE FUNCTION update\_last\_listened\_date() RETURNS TRIGGER AS $$

BEGIN

UPDATE Track SET last\_listened\_date = NEW.listening\_date WHERE id = NEW.track\_id;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER last\_listened\_date\_trigger

AFTER INSERT ON Listening\_History

FOR EACH ROW EXECUTE PROCEDURE update\_last\_listened\_date();

**4.2.3** Триггер для обновления количества песен в плейлисте

Этот триггер автоматически обновляет количество песен в плейлисте при добавлении или удалении песни. Код данного триггера:

CREATE OR REPLACE FUNCTION update\_last\_listened\_date() RETURNS TRIGGER AS $$

BEGIN

UPDATE Track SET last\_listened\_date = NEW.listening\_date WHERE id = NEW.track\_id;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER last\_listened\_date\_trigger

AFTER INSERT ON Listening\_History

FOR EACH ROW EXECUTE PROCEDURE update\_last\_listened\_date();

**4.2.4** Триггер для обновления даты последнего изменения плейлиста

Этот триггер автоматически обновляет дату последнего изменения плейлиста при добавлении или удалении трека. Код данного триггера:

CREATE OR REPLACE FUNCTION update\_last\_modified\_date() RETURNS TRIGGER AS $$

BEGIN

UPDATE Playlist SET last\_modified\_date = NOW() WHERE id = NEW.playlist\_id;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER last\_modified\_date\_trigger

AFTER INSERT OR DELETE ON Playlist\_Track

FOR EACH ROW EXECUTE PROCEDURE update\_last\_modified\_date();

**4.2.5** Триггер для автоматического обновления среднего рейтинга трека

Этот триггер автоматически обновляет средний рейтинг трека при добавлении или изменении оценки. Код данного триггера:

CREATE OR REPLACE FUNCTION update\_average\_rating() RETURNS TRIGGER AS $$

BEGIN

UPDATE Track SET average\_rating = (SELECT AVG(score) FROM Rating WHERE track\_id = NEW.track\_id) WHERE id = NEW.track\_id;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER average\_rating\_trigger

AFTER INSERT OR UPDATE ON Rating

FOR EACH ROW EXECUTE PROCEDURE update\_average\_rating();

**4.2.6** Триггер для удаления треков из плейлистов при удалении трека

Этот триггер автоматически удаляет все записи из таблицы Playlist\_Track, связанные с удаляемым треком. Код данного триггера:

CREATE OR REPLACE FUNCTION delete\_track\_from\_playlists() RETURNS TRIGGER AS $$

BEGIN

DELETE FROM Playlist\_Track WHERE track\_id = OLD.id;

RETURN OLD;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER delete\_track\_trigger

BEFORE DELETE ON Track

FOR EACH ROW EXECUTE PROCEDURE delete\_track\_from\_playlists();

**4.2.7** Триггер для обновления количества подписчиков исполнителя

Этот триггер автоматически обновляет количество подписчиков исполнителя при добавлении или удалении подписки на исполнителя. Код данного триггера:

CREATE OR REPLACE FUNCTION update\_subscriber\_count() RETURNS TRIGGER AS $$

BEGIN

UPDATE Artist SET subscriber\_count = (SELECT COUNT(\*) FROM Artist\_Subscriber WHERE artist\_id = NEW.artist\_id) WHERE id = NEW.artist\_id;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER subscriber\_count\_trigger

AFTER INSERT OR DELETE ON Artist\_Subscriber

FOR EACH ROW EXECUTE PROCEDURE update\_subscriber\_count();

**4.2.8** Триггер для удаления подписок при удалении исполнителя

Этот триггер автоматически удаляет все записи из таблицы Artist\_Subscriber, связанные с удаляемым исполнителем. Код данного триггера:

CREATE OR REPLACE FUNCTION delete\_artist\_subscriptions() RETURNS TRIGGER AS $$

BEGIN

DELETE FROM Artist\_Subscriber WHERE artist\_id = OLD.id;

RETURN OLD;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER delete\_artist\_trigger

BEFORE DELETE ON Artist

FOR EACH ROW EXECUTE PROCEDURE delete\_artist\_subscriptions();

#### 4.3 Создание индексов

Индексы необходимо наложить на наиболее часто используемых атрибутах отношений. Код индексов выглядит следующим образом:

CREATE INDEX idx\_user\_email ON users (email);

CREATE INDEX idx\_track\_title ON Track (title);

CREATE INDEX idx\_artist\_name ON Artist (name);

CREATE INDEX idx\_album\_title ON Album (title);

CREATE INDEX idx\_genre\_title ON Genre (title);

#### 5 ТЕСТИРОВАНИЕ БАЗЫ ДАННЫХ

Необходимо проверить работоспособность составленных триггеров. Для этого составлен простой тестовый сценарий в котором создается композиция, принадлежащая некоторому исполнителю, ассоциируется с плейлистом, а потом удаляется. Таким образом будет установленно, адекватно ли триггеры исполняют свои функции.

Для проверки функций созданы сценарии с работой с плейлистами пользователя.

**5.1 Проверка триггеров**

Добавим необходимые данные в таблицу. Создадим тестового пользователя. На рисунке 5.1 показана запись тестового пользователя.

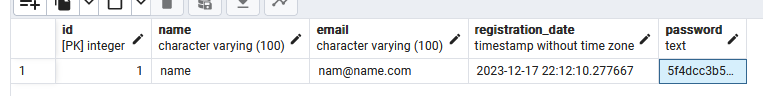


Рисунок 5.1 — Запись тестового пользователя

Создадим тестового исполнителя. На рисунке 5.2 показана запись тестового исполнителя.

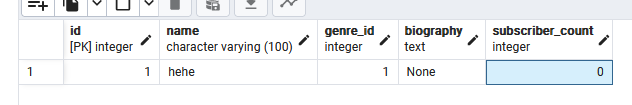


Рисунок 5.2 — Запись тестового исполнителя

Создадим тестовый жанр. На рисунке 5.3 показана запись тестового жанра.

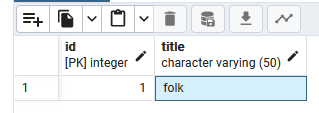


Рисунок 5.3 — Запись тестового жанра

Создадим тестовую композицию. На рисунке 5.4 показана запись тестовой композиции.

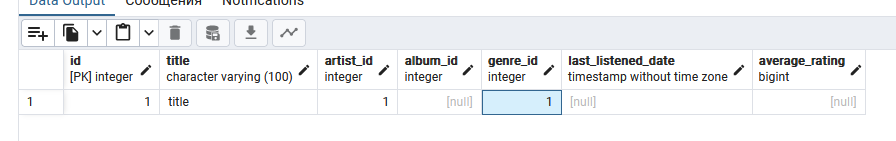


Рисунок 5.4 — Запись тестовой композиции

И наконец создадим тестовый плейлист. На рисунке 5.5 показана запись тестового плейлиста.

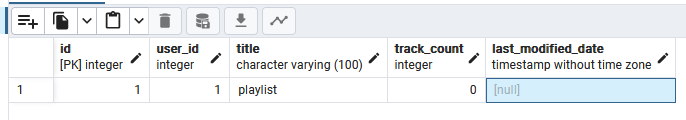


Рисунок 5.5 — Запись тестового плейлиста

В ходе первого этапа проведения теста была построена ассоциация тестовой композиции и тестового плейлиста. Запись, добавленная в отношение playlist\_track приведена на рисунке 5.6.

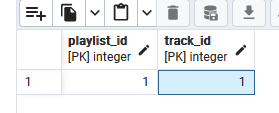


Рисунок 5.6 — Запись об ассоциации композиции и плейлиста

Для проверки срабатывания триггеров необходимо получить запись о текущем состоянии плейлиста. Запись представлена на рисунке 5.7.

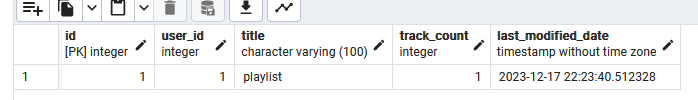


Рисунок 5.6 — Запись о текущем состоянии плейлиста

Тест подтвердил корректность работы первой группы триггеров. Во второй фазу тестирования необходимо проверить, исчезнет ли ассоциация при удалении композиции из базы. На рисунке 5.7 представлена выборка из ассоциативной таблицы после удаления композиции из базы данных. Как видно, ассоциация пропала вместе с ней.

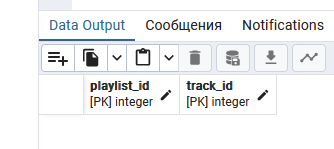


Рисунок 5.7 — Выборка из ассоциативной таблицы

**5.2 Проверка функций**

Для проверки работоспособности функций необходимо добавить в базу данных новые тестовые данные. Новые тестовые композиции представлены на рисунке 5.8.

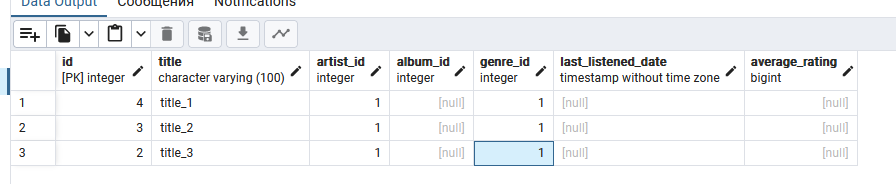


Рисунок 5.8 — Тестовые композиции

Кроме этого созданы новые ассоциативные связи между композициями и плейлистом. Ассоциации между новыми композициями и плейлистом представлены на рисунке 5.9.

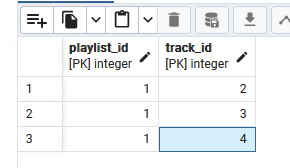


Рисунок 5.9 — Ассоциации между композициями и плейлистом

Выполним функцию count\_tracks\_in\_playlist для получения количества ассоциативных связей между, имеющихся у заданного плейлиста. Результат выполнения функции представлен на рисунке 5.10.

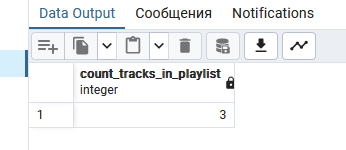


Рисунок 5.10 — Результат выполнения функции

Результат выполнения функции возврата всех плейлистов пользователя get\_user\_playlists представлена на рисунке 5.11.

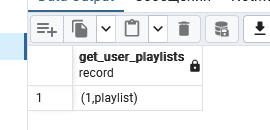


Рисунок 5.11 — Результат выполнения функции

Тесты продемонстрировали нормальную работу функций, написанных для базы данных.

#### ЗАКЛЮЧЕНИЕ

В ходе выполнения курсового проекта была спроектирована и создана база данных для музыкального сервиса.

Кроме этого были изучены литературные источники и проведен сравнительный анализ разных технологий построения баз данных. После анализа наиболее подходящей СУБД для выполнения курсового проекта была выбрана PostgreSQL.

Согласно разработанным схемам и в соответствии с принятыми требованиями была построена база данных, а также разработаны триггеры, индексы и хранимые функции для работы с ней.

#### СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

[1] Против гигантов: Как Spotify подвинул Apple и изменил музыкальную индустрию / Свен Карлссон, Юнас Лейонхуфвуд. — М.: Альпина Паблишер, 2020. — 320 с.

[2] Введение в системы баз данных / Дейт Кристофер Дж. — М.: Издательский дом «Вильямс», 2005. – 1328 с.

[3] SQL: Быстрое погружение / Шилдс Уолтер. — СПб.: Питер, 2022. — 224 с.

[4] SQL / Фиайли Крис — М.: ДМК Пресс, 2008. — 456 с.

[5] PostgreSQL 11. Мастерство разработки / Шениг Ганс-Юрген — М.: ДМК Пресс, 2019. — 352 с.

[6] PostgreSQL. Основы языка SQL: учеб. пособие / Е. П. Моргунов. — СПб.: БХВ-Петербург, 2018. — 336 с.

[7] PostgreSQL изнутри / Е. В. Рогов. — М.: ДМК Пресс, 2022. — 660 с.

[8] NoSQL: новая методология разработки нереляционных баз данных / Фаулер Мартин, Садаладж Прамодкумар Дж. — М.: Издательский дом «Вильямс», 2013. — 192 с.

[9] MongoDB в действии / Бэнкер Кайл. — М.: ДМК Пресс, 2012. — 394 с.

[10] Семь баз данных за семь недель. Введение в современные базы данных и идеологию NoSQL. Под редакцией Жаклин Картер / Редмонд Эрик, Уилсон Джим. Р. — М.: ДМК Пресс, 2013. — 384 с.

[11] Графовые базы данных: новые возможности для работы со связанными данными / Робинсон Ян, Вебер Джим, Эифрем Эмиль. – М.: ДМК Пресс, 2016. – 256 с.

#### ПРИЛОЖЕНИЕ А

#### (обязательное)

#### Листинг кода

CREATE TABLE users (

id SERIAL PRIMARY KEY,

name VARCHAR(100),

email VARCHAR(100),

registration\_date TIMESTAMP

);

CREATE TABLE Playlist (

id SERIAL PRIMARY KEY,

user\_id INT,

title VARCHAR(100),

FOREIGN KEY (user\_id) REFERENCES users (id)

);

CREATE TABLE Track (

id SERIAL PRIMARY KEY,

title VARCHAR(100),

artist\_id INT,

album\_id INT,

genre\_id INT

);

CREATE TABLE Album (

id SERIAL PRIMARY KEY,

title VARCHAR(100),

artist\_id INT

);

CREATE TABLE Artist (

id SERIAL PRIMARY KEY,

name VARCHAR(100),

genre\_id INT,

biography TEXT

);

CREATE TABLE Genre (

id SERIAL PRIMARY KEY,

title VARCHAR(50)

);

CREATE TABLE Subscription (

id SERIAL PRIMARY KEY,

user\_id INT,

type VARCHAR(50),

cost DECIMAL(5,2),

start\_date TIMESTAMP,

end\_date TIMESTAMP,

FOREIGN KEY (user\_id) REFERENCES users (id)

);

CREATE TABLE Listening\_History (

id SERIAL PRIMARY KEY,

user\_id INT,

track\_id INT,

listening\_date TIMESTAMP,

FOREIGN KEY (user\_id) REFERENCES users (id),

FOREIGN KEY (track\_id) REFERENCES Track (id)

);

CREATE TABLE Rating (

id SERIAL PRIMARY KEY,

user\_id INT,

track\_id INT,

score INT,

FOREIGN KEY (user\_id) REFERENCES users (id),

FOREIGN KEY (track\_id) REFERENCES Track (id)

);

CREATE TABLE Review (

id SERIAL PRIMARY KEY,

user\_id INT,

track\_id INT,

text TEXT,

FOREIGN KEY (user\_id) REFERENCES users (id),

FOREIGN KEY (track\_id) REFERENCES Track (id)

);

-- Add the remaining tables similarly

CREATE TABLE Content\_Rights (

id SERIAL PRIMARY KEY,

track\_id INT,

rights\_holder VARCHAR(100),

rights\_type VARCHAR(50),

start\_date TIMESTAMP,

end\_date TIMESTAMP,

FOREIGN KEY (track\_id) REFERENCES Track (id)

);

CREATE TABLE Advertisement (

id SERIAL PRIMARY KEY,

ad\_name VARCHAR(100),

start\_date TIMESTAMP,

end\_date TIMESTAMP,

cost DECIMAL(5,2)

);

CREATE TABLE Listening\_Statistics (

id SERIAL PRIMARY KEY,

user\_id INT,

track\_id INT,

listen\_count INT,

FOREIGN KEY (user\_id) REFERENCES users (id),

FOREIGN KEY (track\_id) REFERENCES Track (id)

);

CREATE TABLE Recommendation (

id SERIAL PRIMARY KEY,

user\_id INT,

track\_id INT,

recommendation\_date TIMESTAMP,

FOREIGN KEY (user\_id) REFERENCES users (id),

FOREIGN KEY (track\_id) REFERENCES Track (id)

);

CREATE TABLE Payment (

id SERIAL PRIMARY KEY,

user\_id INT,

amount DECIMAL(5,2),

payment\_date TIMESTAMP,

FOREIGN KEY (user\_id) REFERENCES users (id)

);

CREATE TABLE Events (

id SERIAL PRIMARY KEY,

name VARCHAR(100),

location VARCHAR(100),

event\_date TIMESTAMP

);

CREATE TABLE Podcast (

id SERIAL PRIMARY KEY,

title VARCHAR(100),

description TEXT,

release\_date TIMESTAMP

);

CREATE TABLE Radio\_Station (

id SERIAL PRIMARY KEY,

name VARCHAR(100),

genre\_id INT,

FOREIGN KEY (genre\_id) REFERENCES Genre (id)

);

CREATE TABLE Chart (

id SERIAL PRIMARY KEY,

name VARCHAR(100),

creation\_date TIMESTAMP

);

CREATE TABLE User\_Settings (

id SERIAL PRIMARY KEY,

user\_id INT,

music\_preferences TEXT,

notification\_settings TEXT,

FOREIGN KEY (user\_id) REFERENCES users (id)

);

CREATE TABLE Notifications (

id SERIAL PRIMARY KEY,

user\_id INT,

notification\_text TEXT,

notification\_date TIMESTAMP,

FOREIGN KEY (user\_id) REFERENCES users (id)

);

CREATE TABLE Social\_Media (

id SERIAL PRIMARY KEY,

user\_id INT,

platform\_name VARCHAR(50),

profile\_link VARCHAR(255),

FOREIGN KEY (user\_id) REFERENCES users (id)

);

CREATE TABLE Gift\_Cards\_and\_Codes (

id SERIAL PRIMARY KEY,

code VARCHAR(50),

value DECIMAL(5,2),

expiration\_date TIMESTAMP

);

CREATE TABLE Support (

id SERIAL PRIMARY KEY,

user\_id INT,

request\_text TEXT,

request\_date TIMESTAMP,

response\_text TEXT,

response\_date TIMESTAMP,

FOREIGN KEY (user\_id) REFERENCES users (id)

);

CREATE TABLE SecurityInfo (

id SERIAL PRIMARY KEY,

user\_id INT,

login\_data TEXT,

login\_history TEXT,

FOREIGN KEY (user\_id) REFERENCES users (id)

);

CREATE TABLE Playlist\_Track (

playlist\_id INT,

track\_id INT,

PRIMARY KEY (playlist\_id, track\_id),

FOREIGN KEY (playlist\_id) REFERENCES Playlist (id),

FOREIGN KEY (track\_id) REFERENCES Track (id)

);

CREATE TABLE Artist\_Subscriber (

artist\_id INT,

user\_id INT,

PRIMARY KEY (artist\_id, user\_id),

FOREIGN KEY (artist\_id) REFERENCES Artist (id),

FOREIGN KEY (user\_id) REFERENCES users (id)

);

CREATE INDEX idx\_user\_email ON users (email);

CREATE INDEX idx\_track\_title ON Track (title);

CREATE INDEX idx\_artist\_name ON Artist (name);

CREATE INDEX idx\_album\_title ON Album (title);

CREATE INDEX idx\_genre\_title ON Genre (title);

CREATE OR REPLACE FUNCTION check\_email\_duplicates() RETURNS TRIGGER AS $$

BEGIN

IF EXISTS (SELECT 1 FROM users WHERE email = NEW.email) THEN

RAISE EXCEPTION 'Пользователь с email % уже существует', NEW.email;

END IF;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER email\_trigger

BEFORE INSERT OR UPDATE ON users

FOR EACH ROW EXECUTE PROCEDURE check\_email\_duplicates();

CREATE OR REPLACE FUNCTION count\_tracks\_in\_playlist(p\_playlist\_id INT) RETURNS INT AS $$

DECLARE

track\_count INT;

BEGIN

SELECT COUNT(\*) INTO track\_count FROM Playlist\_Track WHERE playlist\_id = p\_playlist\_id;

RETURN track\_count;

END;

$$ LANGUAGE plpgsql;

SELECT count\_tracks\_in\_playlist(1);

CREATE OR REPLACE FUNCTION update\_last\_listened\_date() RETURNS TRIGGER AS $$

BEGIN

UPDATE Track SET last\_listened\_date = NEW.listening\_date WHERE id = NEW.track\_id;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER last\_listened\_date\_trigger

AFTER INSERT ON Listening\_History

FOR EACH ROW EXECUTE PROCEDURE update\_last\_listened\_date();

CREATE OR REPLACE FUNCTION update\_track\_count() RETURNS TRIGGER AS $$

BEGIN

UPDATE Playlist SET track\_count = (SELECT COUNT(\*) FROM Playlist\_Track WHERE playlist\_id = NEW.playlist\_id) WHERE id = NEW.playlist\_id;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER track\_count\_trigger

AFTER INSERT OR DELETE ON Playlist\_Track

FOR EACH ROW EXECUTE PROCEDURE update\_track\_count();

CREATE OR REPLACE FUNCTION get\_user\_playlists(p\_user\_id INT) RETURNS TABLE(playlist\_id INT, title VARCHAR(100)) AS $$

BEGIN

RETURN QUERY SELECT id, Playlist.title FROM Playlist WHERE user\_id = p\_user\_id;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION update\_last\_modified\_date() RETURNS TRIGGER AS $$

BEGIN

UPDATE Playlist SET last\_modified\_date = NOW() WHERE id = NEW.playlist\_id;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER last\_modified\_date\_trigger

AFTER INSERT OR DELETE ON Playlist\_Track

FOR EACH ROW EXECUTE PROCEDURE update\_last\_modified\_date();

CREATE OR REPLACE FUNCTION update\_average\_rating() RETURNS TRIGGER AS $$

BEGIN

UPDATE Track SET average\_rating = (SELECT AVG(score) FROM Rating WHERE track\_id = NEW.track\_id) WHERE id = NEW.track\_id;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER average\_rating\_trigger

AFTER INSERT OR UPDATE ON Rating

FOR EACH ROW EXECUTE PROCEDURE update\_average\_rating();

CREATE OR REPLACE FUNCTION get\_top\_tracks\_by\_genre(p\_genre\_id INT) RETURNS TABLE(track\_id INT, title VARCHAR(100), average\_rating DECIMAL(3,2)) AS $$

BEGIN

RETURN QUERY SELECT id, title, average\_rating FROM Track WHERE genre\_id = p\_genre\_id ORDER BY average\_rating DESC LIMIT 10;

END;

$$ LANGUAGE plpgsql;

SELECT \* FROM get\_top\_tracks\_by\_genre(1);

CREATE OR REPLACE FUNCTION delete\_track\_from\_playlists() RETURNS TRIGGER AS $$

BEGIN

DELETE FROM Playlist\_Track WHERE track\_id = OLD.id;

RETURN OLD;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER delete\_track\_trigger

BEFORE DELETE ON Track

FOR EACH ROW EXECUTE PROCEDURE delete\_track\_from\_playlists();

CREATE OR REPLACE FUNCTION update\_subscriber\_count() RETURNS TRIGGER AS $$

BEGIN

UPDATE Artist SET subscriber\_count = (SELECT COUNT(\*) FROM Artist\_Subscriber WHERE artist\_id = NEW.artist\_id) WHERE id = NEW.artist\_id;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER subscriber\_count\_trigger

AFTER INSERT OR DELETE ON Artist\_Subscriber

FOR EACH ROW EXECUTE PROCEDURE update\_subscriber\_count();

CREATE OR REPLACE FUNCTION get\_artist\_albums(p\_artist\_id INT) RETURNS TABLE(album\_id INT, title VARCHAR(100)) AS $$

BEGIN

RETURN QUERY SELECT id, title FROM Album WHERE artist\_id = p\_artist\_id;

END;

$$ LANGUAGE plpgsql;

SELECT \* FROM get\_artist\_albums(1);

CREATE OR REPLACE FUNCTION delete\_artist\_subscriptions() RETURNS TRIGGER AS $$

BEGIN

DELETE FROM Artist\_Subscriber WHERE artist\_id = OLD.id;

RETURN OLD;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER delete\_artist\_trigger

BEFORE DELETE ON Artist

FOR EACH ROW EXECUTE PROCEDURE delete\_artist\_subscriptions();