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

Факультет Информационных технологий

Кафедра Информационных систем и технологий

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

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА КУРСОВОГО ПРОЕКТА**

по дисциплине «Базы данных»

Тема: «Реализация базы данных социальной сети с использованием технологии сравнения SQL и NoSQL решений»

**Исполнитель**

студент 2 курса 7 группы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Н.Д. Володькин

подпись, дата

**Руководитель**

ассистент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Е.И. Комарова

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

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

дата, подпись

Курсовой проект защищен с оценкой

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Е.И. Комарова

подпись дата инициалы и фамилия

Минск 2023

**Оглавление**

[Введение 5](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210422)

[1 Постановка задачи 6](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210423)

[2 Разработка модели базы данных 7](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210424)

[3 Разработка необходимых объектов 11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210425)

[3.1 Табличные пространства 11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210426)

[3.2 Таблицы 11](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210427)

[3.3 Роли 12](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210429)

[3.4 Процедуры и функции 13](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210431)

[3.4.1 Выборка данных из таблиц 14](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210432)

[3.4.2 Добавление данных в таблицы 14](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210433)

[3.4.3 Изменение данных в таблицах 15](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210434)

[3.4.4 Удаление данных из таблиц 16](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210435)

[3.4.5 Заполнение определённых таблиц на 100 000 строк 16](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210437)

[3.5 Триггеры 16](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210441)

[3.6 Представления 17](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210442)

[3.7 Индексы 19](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210442)

[4 Импорт и экспорт в JSON 21](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210443)

[4.1 Импорт данных таблицы в формате JSON 21](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210426)

[4.2 Экспорт данных таблицы в формате JSON 21](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210427)

[5 Сравнение SQL и NoSQL решений 23](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210444)

[5.1 Запросы к базе данных 23](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210426)

[5.2 Определение моделей данных 24](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210427)

[5.3 Использование индексов 26](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210426)

[6 Тестирование производительности 28](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210445)

[7 Краткое описание приложения для демонстрации 30](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210446)

[8 Руководство пользователя 32](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210447)

[Заключение 37](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210448)

[Список использованной литературы 38](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210449)

[Приложение А (Создание таблиц и индексов) 39](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210450)

[Приложение Б (Создание триггеров и инфраструктуры для них) 42](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210450)

[Приложение В (Создание процедур и функций) 45](file:///C:\Users\User\Downloads\Telegram%20Desktop\Записка%20v1.0.docx#_Toc134210451)

**Введение**

Социальные сети - это один из самых популярных и влиятельных видов коммуникации в современном мире. Они позволяют людям обмениваться мнениями, интересами, новостями и развлечениями с миллионами других пользователей. Для социальных сетей очень важно хранение информации, так как она является основой их работы и ценностью для их клиентов. Без надежного и безопасного хранения информации социальные сети не смогут предоставлять свои услуги на высоком уровне и защищать данные своих пользователей от утечек и злоупотреблений.

База данных (бд) является одним из ключевых компонентов для социальной сети, так как она хранит и обрабатывает различные данные о пользователях, сообществах, постах, лайках, комментариях и т.д. Подбор подходящей базы данных и архитектуры - это важный этап разработки любого проекта или приложения, так как от этого зависит эффективность, надежность и безопасность работы с данными.

Современные системы управления базами данных (СУБД) — это комплексы программных средств, которые позволяют создавать, хранить, обрабатывать и предоставлять доступ к большим объемам структурированной или неструктурированной информации. СУБД могут поддерживать различные модели данных, то есть способы представления и организации данных в базе. Реляционные и нереляционные модели данных — это два основных подхода к организации данных в базах данных, которые имеют разные принципы, преимущества и недостатки. Выбор между ними зависит от специфики и играет ключевую роль в проектировании архитектуры бд.

Задачями курсового проекта является проектирование инфраструктуры базы данных, разработка всех необходимых объектов и приложенния, реализация функциональных требований, тестирование производительности.

Функционально должны быть выполнены следующие задачи:

˗ разработка приложения для взаимодействия с БД;

˗ регистрация и авторизация пользователей;

˗ обеспечение социальной функциональности;

˗ анализ работы сервиса;

˗ мониторинг сети администратором.

**1 Постановка задачи**

Главная особенность социальных сетей ˗ огромное количество сложной и взаимосвязанной информации, создаваемой пользователями информации. Для поддержания работоспособности и удобства использования приложения очень важно разработать надежную архитектуру бд.

Целью данного курсового проекта является создание программного обеспечения для социальной сети.

Задача проекта: разработка архитектуры приложения, создание архитектуры базы данных и интерфейса, выполнение тестирования готового продукта.

Функционально должны быть выполнены следующие задачи:

˗ разработка приложения для взаимодействия с БД;

˗ регистрация и авторизация пользователей;

˗ обеспечение социальной функциональности;

˗ анализ работы сервиса;

˗ мониторинг сети администратором.

Должны быть выполнены следующие требования:

˗ база данных должна быть спроектирована в СУБД PostgreSQL.

˗ доступ к данным должен осуществляться только через соответствующие процедуры;

˗ должен быть проведен импорт данных из JSON файлов, экспорт данных в формат JSON;

˗ необходимо протестировать производительность базы данных (на таблицах, содержащих не менее 100 000 строк) и внести изменения в структуру в случае необходимости;

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

Серверная часть приложения должна быть реализовано с использованием фреймворка ASP.NET Core и языка C#, клиентская часть должна быть реализована с использованием фреймворка React и языка JavaScript.

Разрабатываемое приложение должно содержать форму авторизации, ленту постов, страницу профиля и подписок, панель модератора. Пользователь может подписываться на других пользователей, создавать посты, комментировать, оставлять лайки, делиться публикациями и оставлять жалобы. Модератор может просматривать статистику приложения и посты с жалобами и удалять их, блокировать пользователей.

**2 Разработка модели данных**

Для хранения данных в бд была разработана структура взаимосвязанных таблиц с помощью задания отношений. Отношения в реляционной БД - это логические связи между данными, хранящимися в разных таблицах. Отношения позволяют объединять данные из разных источников и извлекать нужную информацию. Отношения могут быть трех типов: один ко одному, один ко многим и много ко многим. Тип отношения зависит от того, сколько строк в одной таблице соответствует строкам в другой таблице.

Из-за особенностей требований для разработки социальной сети, для реализация функционала требуется большое количество отношений много ко многим, которые будут отображать такие связи как лайки, которые пользователи ставят публикациям.

Диаграмма базы данных, спроектированной в ходе разработки приложения представлена на рисунке 2.1.

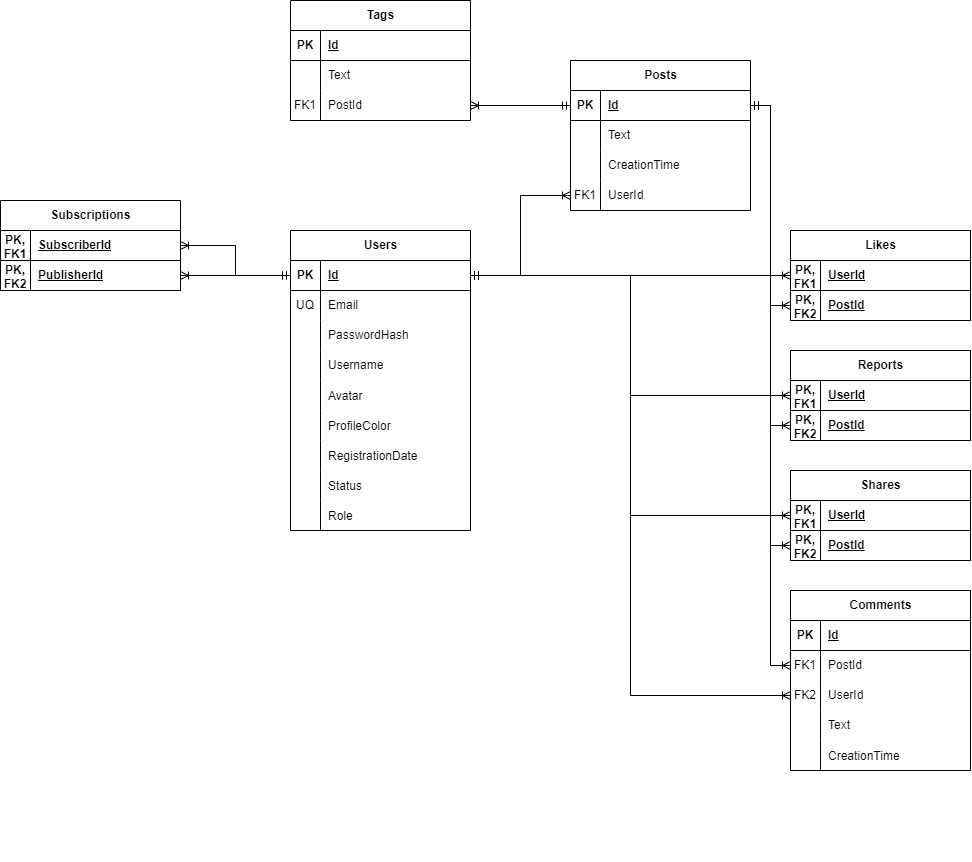


Рисунок 2.1 – Диаграмма базы данных

База данных состоит из 8 таблиц: Users, Posts, Comments, Likes, Shares, Reports, Tags, Subscriptions.

Таблица Students хранит список пользователей, состоит из столбцов (таблица 3.1):

Таблица 3.1 – Столбцы таблицы Users

|  |  |  |
| --- | --- | --- |
| Наименование | Описание | Тип |
| Id | Идентификатор пользователя, первичный ключ | int |
| Email | Почта пользователя, not null, unique | text |
| Username | Псевдоним пользователя, not null | text |
| PasswordHash | Зашифрованный пароль пользователя, not null | text |
| Avatar | Аватар пользователя, not null | text |
| ProfileColor | Цвет оформления профиля пользователя, not null | text |
| RegistrationDate | Дата регистрации пользователя, not null | timestamp with time zone |
| Status | Статус пользователя, not null | text |
| Role | Роль пользователя, not null | text |

Таблица Posts хранит список публикаций, состоит из столбцов (таблица 3.2):

Таблица 3.2 – Столбцы таблицы Posts

|  |  |  |
| --- | --- | --- |
| Наименование | Описание | Тип |
| Id | Идентификатор публикации, первичный ключ | int |
| Text | Текст публикации, not null | text |
| CreationTime | Дата создания публикации, not null | timestamp with time zone |
| UserId | Идентификатор автора публикации, внешний ключ на таблицу Users | int |

Таблица Comments хранит список комментариев к публикациям, состоит из столбцов (таблица 3.3):

Таблица 3.3 – Столбцы таблицы Comments

|  |  |  |
| --- | --- | --- |
| Наименование | Описание | Тип |
| Id | Идентификатор комментраия, первичный ключ | int |
| Text | Текст комментария, not null | text |
| CreationTime | Дата создания публикации, not null | timestamp with time zone |
| UserId | Идентификатор автора комментария, внешний ключ на таблицу Users | int |
| PostId | Идентификатор публикации, которой принадлежит комментарий, внешний ключ на таблицу Posts | int |

Таблица Likes хранит список лайков у публикаций, состоит из столбцов (таблица 3.4):

Таблица 3.4 – Столбцы таблицы Likes

|  |  |  |
| --- | --- | --- |
| Наименование | Описание | Тип |
| UserId | Идентификатор автора лайка, внешний ключ на таблицу Users, часть составного первичного ключа | int |
| PostId | Идентификатор публикации, которой принадлежит лайк, внешний ключ на таблицу Posts, часть составного первичного ключа | int |

Таблица Shares хранит список людей, которые поделились публикацией, состоит из столбцов (таблица 3.5):

Таблица 3.5 – Столбцы таблицы Shares

|  |  |  |
| --- | --- | --- |
| Наименование | Описание | Тип |
| UserId | Идентификатор пользоавтеля, который поделился, внешний ключ на таблицу Users, часть составного первичного ключа | int |
| PostId | Идентификатор публикации, которой поделились, внешний ключ на таблицу Posts, часть составного первичного ключа | int |

Таблица Reports хранит список жалоб на публикацим, состоит из столбцов (таблица 3.6):

Таблица 3.6 – Столбцы таблицы Reports

|  |  |  |
| --- | --- | --- |
| Наименование | Описание | Тип |
| UserId | Идентификатор автора жалобы, внешний ключ на таблицу Users, часть составного первичного ключа | int |
| PostId | Идентификатор публикации, на которую пожаловались, внешний ключ на таблицу Posts, часть составного первичного ключа | int |

Таблица Tags хранит список тэгов публикаций, состоит из столбцов (таблица 3.7):

Таблица 3.7 – Столбцы таблицы Tags

|  |  |  |
| --- | --- | --- |
| Наименование | Описание | Тип |
| Id | Идентификатор тега, первичный ключ | int |
| Text | Наименование тэга, not null | text |
| PostId | Идентификатор публикации, которой принадлежат тэги, внешний ключ на таблицу Posts | int |

Таблица Subscriptions хранит список подписок пользователей, состоит из столбцов (таблица 3.8):

Таблица 3.8 – Столбцы таблицы Subscriptions

|  |  |  |
| --- | --- | --- |
| Наименование | Описание | Тип |
| SubscriberId | Идентификатор подписчика, внешний ключ на таблицу Users, часть составного первичного ключа | int |
| PublisherId | Идентификатор пользоавтеля, на которого подписался подпищик, внешний ключ на таблицу Users, часть составного первичного ключа | int |

**3 Разработка необходимых объектов**

При разработке курсового проекта были созданы следующие объекты: табличные пространства, таблицы, представления, индексы, пользователи, процедуры, триггеры.

3.1 Табличные пространства

Табличное пространство - это концепция, которая позволяет администраторам баз данных определять места в файловой системе, где хранятся файлы, представляющие объекты базы данных, такие как таблицы, индексы и т.д Табличные пространства в PostgreSQL позволяют контролировать расположение файлов на диске и оптимизировать производительность базы данных.

В PostgreSQL Табличные пространства являются неотъемлемой частью кластера базы данных и не могут быть перемещены, подключены или отключены от другого кластера без специальных процедур.

Создание табличного пространства, а также базы данных, которая будет использоавть это пространство представлено на листинге 3.1.1.

CREATE TABLESPACE blurb\_ts

OWNER blurb\_admin

LOCATION '/var/lib/postgresql/data';

CREATE DATABASE blurb\_db

WITH OWNER blurb\_admin

ENCODING 'UTF8'

TABLESPACE blurb\_ts;

Листинг 3.1.1 – Создание табличного пространства и базы данных

3.2 Таблицы

Таблицы в базах данных - это совокупность связанных данных, хранящихся в структурированном виде. Они состоят из столбцов и строк, которые представляют атрибуты и значения объектов базы данных Таблицы позволяют организовать данные логически и избежать избыточности. Для связи таблиц между собой используются ключи - уникальные идентификаторы строк в таблицах.

Для генерации первичных ключей используется IDENTITY. IDENTITY - это свойство столбца в таблице, которое позволяет автоматически генерировать уникальные значения для этого столбца при вставке новых строк. Использование identity упрощает работу с базой данных, так как не нужно заботиться о том, чтобы предоставлять уникальные значения для ключевых столбцов при добавлении данных.

Также для упрощения работы с бд используется опция ON DELETE CASCADE. это опция, которая позволяет автоматически удалять связанные строки из других таблиц при удалении строки из одной таблицы.

Код для создания таблиц представлен в приложении А.

3.3 Роли

Роль - это сущность, которая может владеть объектами и иметь определённые права в базе.

В PostgreSQL нет формального различия между ролями и пользователями. Оба термина используются взаимозаменяемо для обозначения сущностей, которые могут подключаться к базе данных и выполнять действия с ней.

Однако для удобства можно считать, что роль - это более общее понятие, которое включает в себя как индивидуальных пользователей, так и группы пользователей. Пользователь - это роль с атрибутом LOGIN, который позволяет ей аутентифицироваться в системе. Группа - это роль без атрибута LOGIN, которая служит для объединения других ролей по общим правам доступа.

VALID UNTIL означает, что пароль будет действителен до указанной даты и времени.

Создание ролей приведено на листинге 3.3.1.

CREATE ROLE blurb\_admin

WITH LOGIN ENCRYPTED PASSWORD 'SYSTEM'

SUPERUSER CREATEDB CREATEROLE

VALID UNTIL '2023-04-01';

CREATE ROLE blurb\_moderator

WITH LOGIN ENCRYPTED PASSWORD 'SYSTEM'

VALID UNTIL '2023-04-01';

CREATE ROLE blurb\_user

WITH LOGIN ENCRYPTED PASSWORD 'SYSTEM'

VALID UNTIL '2023-04-01';

CREATE ROLE blurb\_server

WITH LOGIN ENCRYPTED PASSWORD 'SYSTEM'

VALID UNTIL '2023-04-01';

CREATE GROUP blurb\_roles;

ALTER GROUP blurb\_roles

ADD USER blurb\_admin, blurb\_moderator, blurb\_user, blurb\_server;

Листинг 3.3.1 – Создание ролей

Привилегии, выдаваемые ролям, определяют, какие действия могут быть выполнены в отношении определенных объектов базы данных (например, таблиц, представлений и т.д.) и контролируют доступ пользователей к этим объектам.

Выдача ролей представлена на листинге 3.3.2.

GRANT ALL

ON "Users", "Likes", "Posts", "Reports", "Shares", "Subscriptions", "Comments", "Tags",

"UserUpdateLogs", "PostUpdateLogs", "CommentUpdateLogs"

TO blurb\_server;

GRANT EXECUTE ON PROCEDURE

add\_user, update\_user, update\_user\_password\_hash, update\_user\_email,

add\_post, update\_post,

add\_comment, update\_comment,

add\_tag, add\_tags,

add\_like, add\_likes, delete\_like, delete\_likes,

add\_report, add\_reports,

add\_share, add\_shares,

add\_subscription, delete\_subscription

TO blurb\_roles;

GRANT EXECUTE ON PROCEDURE

update\_user\_status, delete\_user,

delete\_post, delete\_comment, delete\_tag, delete\_tags,

delete\_report, delete\_reports, delete\_share, delete\_shares

TO blurb\_admin, blurb\_moderator;

GRANT EXECUTE ON PROCEDURE update\_user\_role TO blurb\_admin;

GRANT EXECUTE ON FUNCTION

get\_users\_summary, get\_users\_summary\_by\_registration\_date,

get\_posts\_summary, get\_trending\_posts, get\_reported\_posts,

get\_comments\_on\_post, get\_comments\_from\_user,

get\_subscriptions\_for\_publisher, get\_subscriptions\_of\_user,

get\_post\_likes, get\_user\_likes,

get\_post\_reports, get\_user\_reports,

get\_post\_shares, get\_user\_shares,

get\_post\_tags, get\_popular\_tags

TO blurb\_roles;

Листинг 3.3.2 – Выдача привелегий для ролей

Все роли приложения могут выполнять базовые операции для добавления и обновления данных в таблицах, всем ролям также выдается доступ на функции для выборки, так как они не содержат приватной информации. Ролям администратора и модератора выдаются привилегии на процедуры для удаления данных, а администратор имеет права менять роли пользователей. Специальной роли blurb\_server выдаются привилегии на сами таблицы, что может быть полезно при использовании приложения с ORM и автоматической генерацией SQL кода при разработке с использованием code first approach.

3.4 Процедуры и функции

Процедура в СУБД - это программный объект, блок кода, который может быть вызван многократно, который позволяет выполнять определенные действия с данными в базе данных. Функция в СУБД - это программный объект, который возвращает значение. Процедуры и функции позволяют уменьшить количество кода, а также обеспечить его повторное использование, ускорить запросы к базе данных, ограничить доступ к данным.

* + 1. Выборка данных из таблиц

Процедуры в PostgreSQL не могут возвращать таблицы или результирующме наборы, поэтому для выборки данных используются функции. Всего было разработано 16 функций для выборки различной информации из таблиц и представлений: get\_users\_summary, get\_users\_summary\_by\_registration\_date, get\_posts\_summary, get\_trending\_posts, get\_reported\_posts, get\_comments\_on\_post, get\_comments\_from\_user, get\_subscriptions\_for\_publisher, get\_subscriptions\_of\_user, get\_post\_likes, get\_user\_likes, get\_post\_reports, get\_user\_reports, get\_post\_shares, get\_user\_shares, get\_post\_tags, get\_popular\_tags.

Пример создания функции в PostgreSQL представлен на листинге 3.4.1.1.

CREATE OR REPLACE FUNCTION get\_popular\_tags()

RETURNS TABLE (Id integer, Text text, PostId integer, Usage integer)

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

RETURN QUERY

SELECT "Tags"."Text", COUNT("Tags"."PostId") AS "Usage"

FROM "Tags"

GROUP BY "Tags"."Text"

ORDER BY COUNT("Tags"."PostId") DESC;

END;

$$ LANGUAGE plpgsql;

Листинг 3.4.1.1 – Создание функции get\_popular\_tags

По умолчанию в PostgreSQL функции и процедуры выполняются с привилегиями той роли, которая вызвала процедуры, а это значит, что пользователи не смогут получить доступ к таблице. Чтобы обойти это ограничение, все функции и процедуры были созданы с опцией SECURITY DEFINER, что позволяет вызывать процедуры с теми привилегиями, которые есть у владельца процедуры. Чтобы избежать уязвимостей, связанных с временными таблицами используется свойство search\_path, которое устанавливается в желаемую схему и временную схему по умолчанию.

* + 1. Добавление данных в таблицы

Данные в таблицу будут добавляться, изменяться и удаляться при помощи процедур. Для добавления данных были созданы следующие процедуры: add\_user, add\_post, add\_comment, add\_tag, add\_tags, add\_like, add\_likes, add\_report, add\_reports, add\_share, add\_shares, add\_subscription.

Создание процедуры для добавления тэгов к публикации представлено на листинге 3.4.2.1.

CREATE OR REPLACE PROCEDURE add\_tags(postid integer, tags text[])

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

INSERT INTO "Tags" ("Text", "PostId")

SELECT unnest(tags), postid;

END;

$$

Листинг 3.4.2.2 – Создание процедуры add\_tags

Процедура add\_tags использует возможность PostgreSQL работать с массивами значений, что позволяет за один вызов процедуры добавить несколько строк в таблицу.

* + 1. Изменение данных в таблицах

Процедуры, отвечающие за обновление данных: update\_user, update\_user\_status, update\_user\_role, update\_user\_password\_hash, update\_user\_email, update\_post, update\_comment.

Создание процедуры для обновления профиля представлено на листинге 3.4.3.1.

CREATE OR REPLACE PROCEDURE update\_user(id integer, profile\_color text DEFAULT NULL, avatar text DEFAULT NULL, username text DEFAULT NULL)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

UPDATE "Users"

SET "ProfileColor" = COALESCE(profile\_color, "ProfileColor"),

"Avatar" = COALESCE(avatar, "Avatar"),

"Username" = COALESCE(username, "Username")

WHERE "Id" = id;

END;

$$

Листинг 3.4.3.1 – Создание процедуры update\_user

Процедура update\_user использует значения параметров процедуры по умолчанию и функцию COALESCE, которая выбирает первое значение из набора, которое не является NULL, чтобы дать возможность обновить только определенные настройки профиля.

* + 1. Удаление данных из таблиц

Для удаления данных были разработаны следующие процедуры: delete\_user, delete\_post, delete\_comment, delete\_tag, delete\_tags, delete\_like, delete\_likes, delete\_report, delete\_reports, delete\_share, delete\_shares, delete\_subscription.

Создание процедуры для удаления публикации представлено на листинге 3.4.4.1.

CREATE OR REPLACE PROCEDURE delete\_post(id integer)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

DELETE FROM "Posts"

WHERE "Id" = id;

END;

$$

Листинг 3.4.4.1 – Создание процедуры delete\_post

* + 1. Заполнение определённых таблиц на 100000 строк

На листинге 3.4.5.1 представлен скрипт, заполняющий таблицу Likes более 100000 строками. Генерация случайных чисел позволяет создать различное количество лайков для постов в диапазоне от 1000 до 2000 лайков на пост.

DO $$

DECLARE

i integer;

j integer;

num\_likes integer;

BEGIN

FOR i IN 1..100 LOOP

num\_likes := floor(random() \* (2000 - 1000 + 1) + 1000);

FOR j IN 1..num\_likes LOOP

CALL add\_like(j, i);

END LOOP;

END LOOP;

END;

$$ LANGUAGE plpgsql;

Листинг 3.4.5.1 – Скрипт, заполняющий таблицу Likes

3.5 Триггеры

Триггеры SQL - это триггеры баз данных, которые позволяют вам указать движку SQL выполнить определенный код при наступлении определенного события или даже перед его наступлением. В PostgreSQL исполняемый код описывается путем создания функции, которая возвращает значение типа TRIGGER.

Для обеспечения сохранности данных все операции по редактированию профилей, публикаций и комментариев логгируются с помощью триггеров. Были созданы 3 таблицы, функции и триггера для лоогирования данных

На листинге 3.5.1 представлено создание триггера для логгирования.

CREATE TABLE "UserUpdateLogs" (

"Id" integer GENERATED ALWAYS AS IDENTITY,

"UserId" integer NOT NULL,

"UpdatedColumns" text[] NOT NULL,

"OldValues" text[] NOT NULL,

"NewValues" text[] NOT NULL,

"UpdateTimestamp" timestamp with time zone NOT NULL,

CONSTRAINT "PK\_UserUpdateLogs" PRIMARY KEY ("Id")

);

CREATE OR REPLACE FUNCTION log\_user\_update()

RETURNS TRIGGER AS $$

BEGIN

IF OLD."Email" = NEW."Email" AND OLD."RegistrationDate" = NEW."RegistrationDate" AND OLD."PasswordHash" = NEW."PasswordHash"

THEN

INSERT INTO "UserUpdateLogs" ("UserId", "UpdatedColumns", "OldValues", "NewValues", "UpdateTimestamp")

VALUES (

OLD."Id",

ARRAY(SELECT column\_name FROM information\_schema.columns

WHERE table\_name = 'Users' AND column\_name NOT IN ('Id', 'Email', 'RegistrationDate', 'PasswordHash')),

ARRAY[OLD."Username", OLD."Avatar", OLD."ProfileColor", OLD."Status", OLD."Role"],

ARRAY[NEW."Username", NEW."Avatar", NEW."ProfileColor", NEW."Status", NEW."Role"],

NOW()

);

END IF;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER user\_update\_trigger

AFTER UPDATE ON "Users"

FOR EACH ROW

EXECUTE FUNCTION log\_user\_update();

Листинг 3.5.1 – Создание user\_update\_trigger

* 1. Представления

Представление - это виртуальная таблица, которая представляет собой поименованный запрос, который будет подставлен как подзапрос при использовании представления. В отличие от обычных таблиц, представление не является самостоятельной частью набора данных, хранящихся в базе данных.

Преимуществами представлений являются возможность гибкой настройки данных, а также сокрытии приватной информации. Были разработаны представления для таблицы Users и Posts, которые содержат вычисляемую информацию, а также исключают электронную почту и пароль пользователя.

Создание процедуры для удаления пользователя и публикации представлено на листингах 3.6.1 и 3.6.2.

CREATE OR REPLACE VIEW "UsersSummary" AS

SELECT "Users"."Id", "Users"."Username",

"Users"."Avatar", "Users"."ProfileColor", "Users"."RegistrationDate", "Users"."Status", "Users"."Role",

(SELECT COUNT(\*) FROM "Posts" WHERE "Posts"."UserId" = "Users"."Id") AS "PostCount",

(SELECT COUNT(\*) FROM "Comments" WHERE "Comments"."UserId" = "Users"."Id") AS "CommentCount",

(SELECT COUNT(\*) FROM "Likes" WHERE "Likes"."UserId" = "Users"."Id") AS "LikeCount",

(SELECT COUNT(\*) FROM "Reports" WHERE "Reports"."UserId" = "Users"."Id") AS "ReportCount",

(SELECT COUNT(\*) FROM "Shares" WHERE "Shares"."UserId" = "Users"."Id") AS "ShareCount",

(SELECT COUNT(\*) FROM "Subscriptions" WHERE "Subscriptions"."SubscriberId" = "Users"."Id") AS "SubscriptionCount"

FROM "Users";

Листинг 3.6.1 – Создание процедуры UsersSummary

CREATE OR REPLACE VIEW "UsersSummary" AS

SELECT "Users"."Id", "Users"."Username",

"Users"."Avatar", "Users"."ProfileColor", "Users"."RegistrationDate", "Users"."Status", "Users"."Role",

(SELECT COUNT(\*) FROM "Posts" WHERE "Posts"."UserId" = "Users"."Id") AS "PostCount",

(SELECT COUNT(\*) FROM "Comments" WHERE "Comments"."UserId" = "Users"."Id") AS "CommentCount",

(SELECT COUNT(\*) FROM "Likes" WHERE "Likes"."UserId" = "Users"."Id") AS "LikeCount",

(SELECT COUNT(\*) FROM "Reports" WHERE "Reports"."UserId" = "Users"."Id") AS "ReportCount",

(SELECT COUNT(\*) FROM "Shares" WHERE "Shares"."UserId" = "Users"."Id") AS "ShareCount",

(SELECT COUNT(\*) FROM "Subscriptions" WHERE "Subscriptions"."SubscriberId" = "Users"."Id") AS "SubscriptionCount"

FROM "Users";

Листинг 3.6 .2 – Создание процедуры PostsSummary

* 1. Индексы

Индекс в базе данных - это структура данных, которая улучшает скорость поиска данных в таблице. Индексы создаются для одного или нескольких столбцов таблицы и обеспечивают быстрый доступ к строкам таблицы, которые соответствуют определенным критериям поиска. Индексы могут существенно ускорить выполнение запросов к базе данных, но также могут замедлить операции вставки, обновления и удаления, так как индексы должны быть обновлены при каждой такой операции.

PostgreSQL предоставляет несколько типов индексов: B-Tree, Hash, GiST, SP-GiST, GIN и BRIN. Каждый тип индекса использует разные алгоритмы, которые лучше всего подходят для разных типов запросов1.

B-Tree: B-Tree индексы могут обрабатывать запросы на равенство и диапазон для данных, которые могут быть отсортированы в некотором порядке. Оптимизатор запросов PostgreSQL будет рассматривать использование B-Tree индекса всякий раз, когда индексируемый столбец участвует в сравнении с использованием одного из этих операторов: <, <=, =, >= и >1.

Hash: Hash индексы хранят 32-битный хеш-код, полученный из значения индексируемого столбца. Следовательно, такие индексы могут обрабатывать только простые сравнения на равенство1.

GiST: GiST (Generalized Search Tree) индексы не являются одним видом индекса, а скорее инфраструктурой, в рамках которой могут быть реализованы многие различные стратегии индексации. Соответственно, конкретные операторы, с которыми может быть использован GiST индекс, варьируются в зависимости от стратегии индексации (класса оператора)1.

SP-GiST: SP-GiST (Space-Partitioned GiST) индексы предоставляют инфраструктуру для создания масштабируемых и эффективных по памяти некластеризованных индексов1.

GIN: GIN (Generalized Inverted Index) индексы оптимизированы для обработки запросов с операторами @>, <@ и &&1.

BRIN: BRIN (Block Range Index) индексы хранят метаданные о диапазонах страниц данных в таблице и могут обеспечить значительное ускорение запросов для очень больших таблиц.

По умолчанию используются B-Tree индексы, потому что они подходят для большинства задач. Часто столбцов со строковыми данными может быть использован Hash индекс, но они подходят только для операций типа =, что неподходит под требования для созданных таблиц.

Создание индексов представлено на листинге 3.7.1.

CREATE INDEX "IX\_Users\_RegistrationDate" ON "Users" ("RegistrationDate");

CREATE INDEX "IX\_Posts\_CreationTime" ON "Posts" ("CreationTime");

CREATE INDEX "IX\_Posts\_UserId" ON "Posts" ("UserId");

CREATE INDEX "IX\_Comments\_CreationTime" ON "Comments" ("CreationTime");

CREATE INDEX "IX\_Comments\_UserId" ON "Comments" ("UserId");

CREATE INDEX "IX\_Comments\_PostId" ON "Comments" ("PostId");

CREATE INDEX "IX\_Likes\_UserId" ON "Likes" ("UserId");

CREATE INDEX "IX\_Likes\_PostId" ON "Likes" ("PostId");

CREATE INDEX "IX\_Reports\_UserId" ON "Reports" ("UserId");

CREATE INDEX "IX\_Reports\_PostId" ON "Reports" ("PostId");

CREATE INDEX "IX\_Shares\_UserId" ON "Shares" ("UserId");

CREATE INDEX "IX\_Shares\_PostId" ON "Shares" ("PostId");

CREATE INDEX "IX\_Subscriptions\_SubscriberId" ON "Subscriptions" ("SubscriberId");

CREATE INDEX "IX\_Subscriptions\_PublisherId" ON "Subscriptions" ("PublisherId");

CREATE INDEX "IX\_Tags\_PostId" ON "Tags" ("PostId");

CREATE INDEX "IX\_Tags\_Text" ON "Tags" ("Text");

Листинг 3.7.1 – Создание индексов

**4 Импорт и экспорт в JSON**

4.1 Импорт данных таблицы в формате JSON

JSON (JavaScript Object Notation) - это стандартный текстовый формат для представления структурированных данных на основе синтаксиса объекта JavaScript. JSON является одним из наиболее популярных форматов для хранения данных из—за его широкого применения в веб-разработке, простоте в использовании, гибкости и возможности чтения человеком.

На листинге 4.1.1 представлена процедура, выполняющая вставку данных в таблицу Posts из формата JSON.

CREATE OR REPLACE PROCEDURE import\_posts\_json(path\_to\_file text)

LANGUAGE plpgsql

AS $$

BEGIN

CREATE TEMP TABLE temp (data jsonb);

EXECUTE format('COPY temp (data) FROM %L', path\_to\_file);

INSERT INTO "Posts" ("Text", "CreationTime", "UserId")

SELECT data->>'Text', (data->>'CreationTime')::timestamp with time zone, (data->>'UserId')::integer

FROM temp;

DROP TABLE temp;

END;

$$;

Листинг 4.1.1 – Процедура, выполняющая импорт данных в формате JSON

Чтобы сохранить данные JSON документа в таблице в PostgreSQL можно использовать тип данных JSON или JSONB. JSONB хранит документ в бинарном формате и поддерживает индексирование, JSON же хранит данные в текстовом виде и не поддерживает индексирование. По скольку поля в типе данных JSON и JSONB все имеют тип данных text, их нужно явно привести к изначальным типам данных столбцов таблицы при импорте.

* 1. Экспорт данных таблицы в формате JSON

На листинге 4.2.1 представлена процедура, выполняющая экспорт данных из таблицы Posts в формат JSON. PostgreSQL предоставляет встроенный функционал для импорта данных в JSON через функцию COPY.

CREATE OR REPLACE PROCEDURE export\_posts\_json(path\_to\_file text)

LANGUAGE plpgsql

AS $$

BEGIN

EXECUTE format('COPY (SELECT row\_to\_json(t) FROM (SELECT \* FROM "Posts") t) TO %L', path\_to\_file);

END;

Листинг 4.2.1 – Процедура, выполняющая экспорт данных в формате JSON

**5 Сравнение SQL и NoSQL решений**

NoSQL - это подход к проектированию баз данных, который позволяет хранить и запрашивать данные за пределами традиционных структур, используемых в реляционных базах данных. Одним из самых популярных NoSQL решений сегодня является MongoDB.

MongoDB является документо-ориентированной системой, в которой центральным понятием является документ. Документ можно представить как объект, хранящий некоторую информацию. Документная база данных – это тип нереляционных баз данных, предназначенный для хранения и запроса данных в виде документов в формате, подобном JSON. Гибкий, полуструктурированный, иерархический характер документов и документных баз данных позволяет им развиваться в соответствии с потребностями приложений.

5.1 Запросы к базе данных

В реляционных базах данных для запросов используется язык SQL. SQL (Structured Query Language) - это стандартизированный язык для добавления, доступа, обновления и удаления данных в реляционных базах данных.

Однако для использования данных в приложении требуется решение для преобразования данных из таблиц реляционной бд в объекты языка программирования. Сегодня для решения данной проблемы используются ORM. ORM (Object-Relational Mapping) - это фреймворк, который может помочь и упростить перевод между двумя парадигмами: объектами и таблицами реляционных баз данных. Он может использовать определения классов (модели) для создания, поддержки и обеспечения полного доступа к данным объектов и их сохранности в базе данных.

Большинство ORM, например Entity Framework Core, позволяют делать запросы к базе данных используя собственные методы и инструменты языка программирования. Это повышает скорость разработки, однако может усложнить написание более сложных запросов а также очень негативно сказаться на производительности.

Для решения данной проблемы можно использовать инструменты фреймворка для получения результата SQL запроса, но в таком случае мы будем ограничены моделями нашего приложения, которые обязаны соответствовать результату SQL запроса.

Для запросов к MongoDB используется язык MQL. MQL (MongoDB Query Language) - это гибкий и интуитивно понятный язык для работы с данными в MongoDB. MQL похож на SQL и предназначен для запросов к неструктурированным данным. Запросы в MongoDB основаны на JavaScript.

На практике при работе с MongoDB MQL используется очень редко. MongoDB предоставляет большое количество ORM и драйверов для различных платформ и языков программирования, которые используются для разработки приложений, использующих MongoDB.

На листинге 5.1.1 представлены основные CRUD операции с MongoDB используя MongoDB.Driver для языка C#.

// создание пользователя

User user = new User { Email = "mail@mail.com" };

// добавление пользователя в бд

users.InsertOne(user);

// определение фильтра и поиск документа

var filter = new BsonDocument { { "Email", "mail@mail.com" } };

var userFromDB = users.Find(filter).FirstOrDefault();

// изменение объекта и сохранения изменений в бд

userFromDB.Username = "user";

users.ReplaceOne(filter, userFromDB);

// удаление документа

users.DeleteOne(filter);

Листинг 5.1.1 – CRUD операции с MongoDB

Для добавления данных используются методы InsertOne и InsertMany, также существуют асинхронные версии этих и всех других методов для работы с базой данных. При добавлении документа для него автоматически генерируется уникальный идентификатор типа ObjectId. ObjectId - тип данных, который является частью спецификации BSON, остоит из 12 байт и включает в себя: 4-байтовый временной штамп, представляющий время создания ObjectId в секундах с момента эпохи Unix. 5-байтовое случайное значение, генерируемое один раз для каждого процесса. Это случайное значение уникально для машины и процесса. 3-байтовый счетчик, увеличивающийся на единицу и инициализированный случайным значением.

Для получения данных ипользуется метод find, который принимает в качестве параметра фильтр. Фильтры могут содержать операторы для поиска по столбцам, также поддерживается поиск по регулярным выражениям. Фильтры также используются в методах для обновления и удаления информации.

Различные ORM и драйверы MongoDB схожи и работают также, как и драйвер для C#, что позволяет изучить Mongo один раз и использовать его для проектов на разных языках и фреймворках.

5.2 Определение моделей данных

При определении моделей для реляционных баз данных используется процесс нормализации. Нормализация - это процесс организации данных в базе данных таким образом, чтобы минимизировать избыточность и обеспечить отсутствие многих видов логических ошибок обновления и выборки данных. Важным элементом этого процесса является создание большого количества связанных таблиц для уменьшения избыточности и представления различных отношений между данными.

При использовании документно-ориентированной модели процесс сильно отличается. MongoDB не предоставляет инстрементов для определения схемы данных, таким образом в коллецию можно добавлять документы любой формы и структуры. Однако схему для документа можно определить на стороне приложения, представлено на листингах 5.2.1 и 5.2.2.

public class User

{

[BsonId]

public ObjectId Id { get; set; }

public string Email { get; set; }

public string PasswordHash { get; set; }

public string Username { get; set; }

public string Avatar { get; set; }

public string ProfileColor { get; set; }

public DateTime RegistrationDate { get; set; }

public string Status { get; set; }

public string Role { get; set; }

public List<ObjectId> Subscriptions { get; set; }

}

Листинг 5.2.1 – Модель User

public class Post

{

[BsonId]

public ObjectId Id { get; set; }

public string Text { get; set; }

public DateTime CreationTime { get; set; }

public ObjectId AuthorId { get; set; }

public List<ObjectId> Likes { get; set; }

public List<ObjectId> Shares { get; set; }

public List<ObjectId> Reports { get; set; }

public List<Comment> Comments { get; set; }

public List<string> Tags { get; set; }

public class Comment

{

public ObjectId AuthorId { get; set; }

public string Text { get; set; }

public DateTime CreationTime { get; set; }

}

}

Листинг 5.2.2 – Модель Post

Для хранения дополнительной или связанной информации используются вложенные документы или коллекции вложенных документов. Отношения между документами можно задать используя ссылки на идентификаторы ObjectId связанных документов.

Таким образом при разработке для MongoDB обычно достаточно 1-3 коллеций для представления всех данных.

5.3 Использование индексов

Процесс индексирования схож с реляционными бд. MongoDB поддерживает несколько типов индексов, включая единый индекс (используется для ссылки только на одно поле документа в коллекции), составной индекс (относится к нескольким полям), текстовый индекс (используется для поиска текста в полях), индекс для нескольких ключей (используется для индексации массивов) и хэш-индекс (используется для равномерного распределения данных по шардам).

Единый индекс используется для ссылки только на одно поле документа в коллекции. Составной индекс относится к нескольким полям и может включать до 10 полей 2. Текстовый индекс используется для поиска текста в полях и поддерживает поиск слов и фраз. Индекс для нескольких ключей используется для индексации массивов и может индексировать элементы массива как отдельные ключи. Хэш-индекс используется для равномерного распределения данных по шардам и поддерживает только точечные запросы.

Создание различных индексов в MongoDB представлено на листинге 5.3.1.

var indexOptions = new CreateIndexOptions();

// создание индекса

var signleFieldIndexKeys = Builders<User>.IndexKeys.Ascending(user => user.RegistrationDate);

var singleFieldIndexModel = new CreateIndexModel<User>(signleFieldIndexKeys, indexOptions);

users.Indexes.CreateOne(singleFieldIndexModel);

// создание составного индекса

var compoundIndexKeys = Builders<User>.IndexKeys

.Ascending(user => user.Status)

.Ascending(user => user.Role);

var compoundIndexModel = new CreateIndexModel<User>(signleFieldIndexKeys, indexOptions);

users.Indexes.CreateOne(singleFieldIndexModel);

// создание индекса для нескольких ключей (коллекции)

var multikeyIndexKeys = Builders<User>.IndexKeys.Ascending(user => user.Subscriptions);

var multikeyIndexModel = new CreateIndexModel<User>(multikeyIndexKeys, indexOptions);

users.Indexes.CreateOne(multikeyIndexModel);

// создание текстового индекса

var textIndexKeys = Builders<User>.IndexKeys.Text(user => user.Username);

var textIndexModel = new CreateIndexModel<User>(textIndexKeys, indexOptions);

users.Indexes.CreateOne(textIndexModel);

// создание хэш индекса

var hashIndexKeys = Builders<User>.IndexKeys.Hashed(user => user.Username);

var hashIndexModel = new CreateIndexModel<User>(hashIndexKeys, indexOptions);

users.Indexes.CreateOne(hashIndexModel);

Листинг 5.3.1 – Создание индексов в MongoDB

5.4 Итоги

Таким образом документно-ориентированная модель может быть хорошим решением для задач, в которых не пресутсвует большое количетсво связанных между собой данных, и особенно для хранения слабоструктурированных данных.

Документно-ориентирования модель является хорошим решением для построения социальная сети, однако при расширении требований и нагрузки может потерять свои преимущества, требуя перестроения архитектуры и смены модели данных.

**6 Тестирование производительности**

Производительность базы данных является критически важным фактором для обеспечения быстрого и эффективного доступа к данным. Ключевым в обеспечении произваодительности бд является правильное использование индексов. Правильное использование индексов может значительно улучшить производительность базы данных и обеспечить быстрый и надежный доступ к данным.

Для тестирования производительности была выбрана таблица Likes, так как лайки являются самым часто используемым действием взаимодействия в социальной сети. После заполнения таблицы с используем генерации случайных значений в таблице находится 145335 строк, представлено на рисунке 6.1.

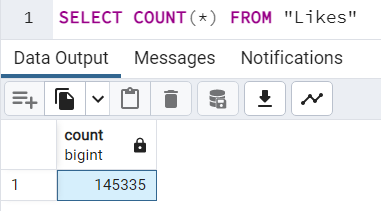


Рисунок 6.1 – Число строк в таблице

Для проверки эффективности использования индексов выполним типичную задачу для подобной таблицы - подсчет количества лайков на определенной публикации и оценим план запроса и его стоимость, представлено на рисунке 6.2 и 6.3.

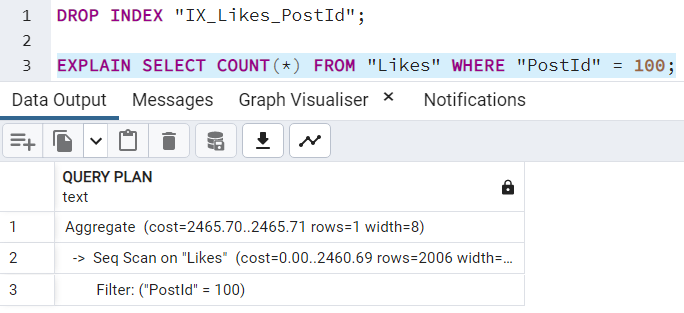


Рисунок 6.2 – План запроса без использования индекса

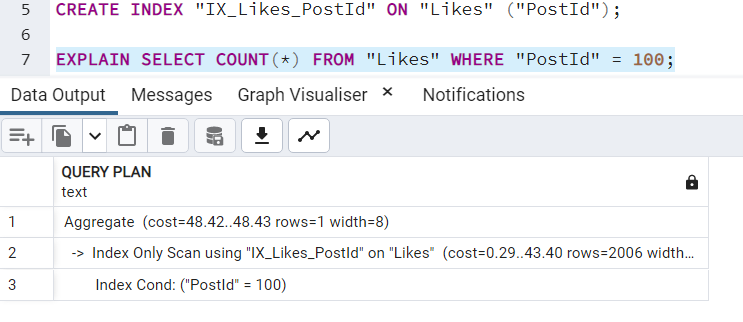


Рисунок 6.3 – План запроса с использованием индекса

Использование индекса существенно понижает стоимость агрегации, что увеличивает производительность запроса.

Также важным является использование составных первичных ключей, что оптимально для запросов где используется фильтрация по обоим столбцам, например при ситуации, когда нужно узнать поставил ли пользователь лайк, представлено на рисунке 6.4.

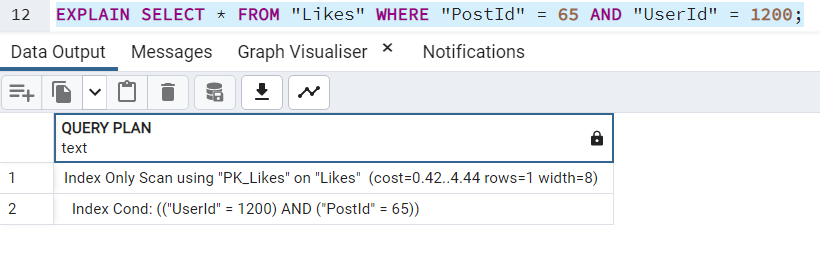


Рисунок 6.4 – Запрос с фильтром по обоим столбцам

**7 Краткое описание приложения для демонстрации**

Для курсового проекта было разработано приложение социальной сети. Приложение состоит из двух частей: серверной и клиентской.

Серверная часть приложения была разработана с использованием ASP.NET Core. ASP.NET Core - это свободно распространяемый кроссплатформенный фреймворк для создания веб-приложений на платформе .NET с открытым исходным кодом. Эта платформа разрабатывается компанией Microsoft совместно с сообществом и имеет большую производительность по сравнению с ASP.NET. Она имеет модульную структуру и совместима с такими операционными системами, как Windows, Linux и macOS.

Существует множество решений для интеграции базы данных в приложение на платформе .NET: ADO.NET, Dapper, Entity Framework Core, для курсового проекта был выбран Entity Framework. Entity Framework (EF) - это объектно-реляционное отображение, которое позволяет разработчикам .NET работать с реляционными данными с помощью объектов, специфичных для предметной области. Он устраняет необходимость в большинстве кода доступа к данным, который обычно необходимо писать разработчикам. Entity Framework позволяет разработчикам писать запросы используя SQL или LINQ.

Приложение построено на двухслойной архитектуре. Первый слой представляет собой слой доступа к данным. Данный слой содержит интерфейсы, предоставляющие собой контракт доступа к данным бд, а также реализации этих интерфейсов - репозитории. Репозиторий - это паттерн проектирования, который является промежуточным звеном между классами, непосредственно взаимодействующими с данными, и остальной программой. Он позволяет абстрагироваться от конкретных подключений к источникам данных, с которыми работает программа.

Второй слой содержит в себе бизнес логику, такую как авторизация пользователей, преобразование данных для потребления клиентом, а также отвечает за представление данных клиенту с помощью контроллеров ASP.NET Core. Для доступа к данным контроллеры потребляют интерфейсы репозиториев с помощью встроенной функциональности Dependency Injection фреймворка ASP.NET Core. Все требуемые зависимости добавляются в качестве параметров конструктора контроллера и автоматически предоставляются фреймворком. Такое решение позволяет облегчить инициализацию всех объектов приложения, а использование интерфейсов позволяет достичь слабосвязанного кода.

Приложение также отвечает за авторизацию и аутентификацию пользователей, хэширование паролей. Авторизация и аутентификация основана встроенном функционале фреймворка с использование Jwt-токенов, а хэширование и верификация паролей реализована с использованием алгоритма Bcrypt.

Клиенсткая часть приложения представляет собой веб-приложение, разработанное с использованием JavaScript фреймворка Next.JS. Next.js - это фреймворк на основе React, созданный Vercel и используемый для создания производительных статических веб-приложений. Next.js предлагает функции рендеринга на стороне клиента и сервера и используется для разработки полноценных приложений React.

В качестве стилевого решения был выбран Tailwind CSS - это постепенно набирающий популярность CSS-фреймворк, позволяющий вносить изменения в оформление сайтов и приложений, не покидая HTML-разметку и не используя тег <style>. Он предлагает обширный каталог классов и инструментов для облегчения стилизации сайта или приложения. В отличие от большинства фреймворков, Tailwind CSS не предоставляет готовые компоненты, а только утилитарные классы, предоставляющие доступ к базовым свойствам CSS, что позволяет достичь максимальной гибкости при стилизации компонентов.

Приложение имеет несколько страниц и предоставляет весь функционал, требуемый для использования социальной сети пользователями и администраторами. Благодаря комбинации рендеринга со стороны сервера и гидрации приложение динамически реагирует на действия пользователя, такие как лайки и комментарии.

**8 Руководство пользователя**

При первоначальном входе на сайт пользователь попадает на страницу регистрации. После ввода данных для регистрации пользователь попадает на страницу входа в приложение, где пользователь может авторизироваться по электронной почте и паролю. Страницу авторизации и регистрации представлены на рисунках 8.1 и 8.2.

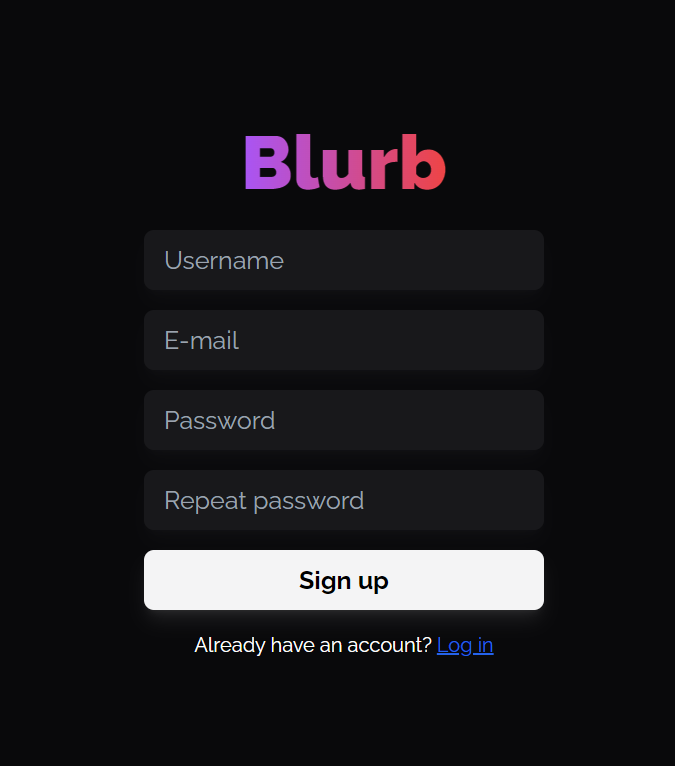


Рисунок 8.1 – Страница регистрации

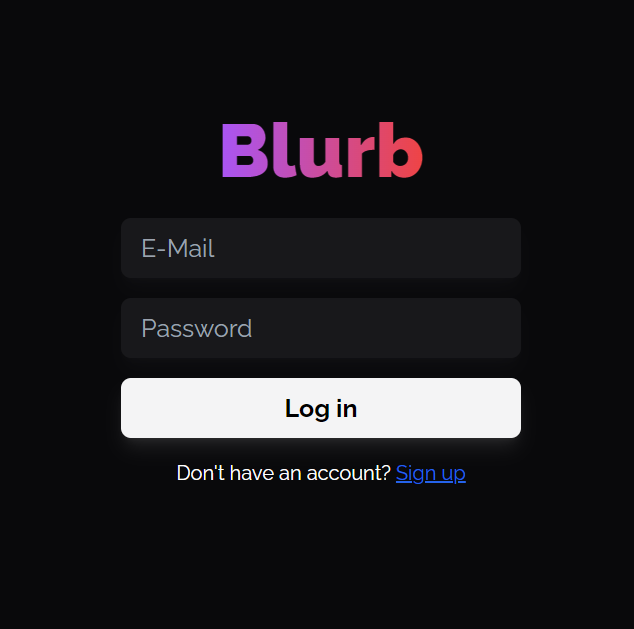


Рисунок 8.2 – Страница авторизации

После авторизации пользователь попадает на страницу ленты новостей, представлено на рисунке 8.3. Здесь пользователь может найти публикацию по тексту или тэгам воспользовавшись полем поиска или создать свою публикацию, написав в текст в соответствующее поле. Пользователь может добавить любой тег к публикации, достаточно лишь написать его в тоже поле, начав слово с символа решетки (#). Под этими полями располагается лента публикаций, где отображаются популярные публикации и публикации авторов, на которых подписан пользователь. С помощью кнопок в хэдере страницы пользователь может перейти на страницу подписок и на страницу своего профиля.

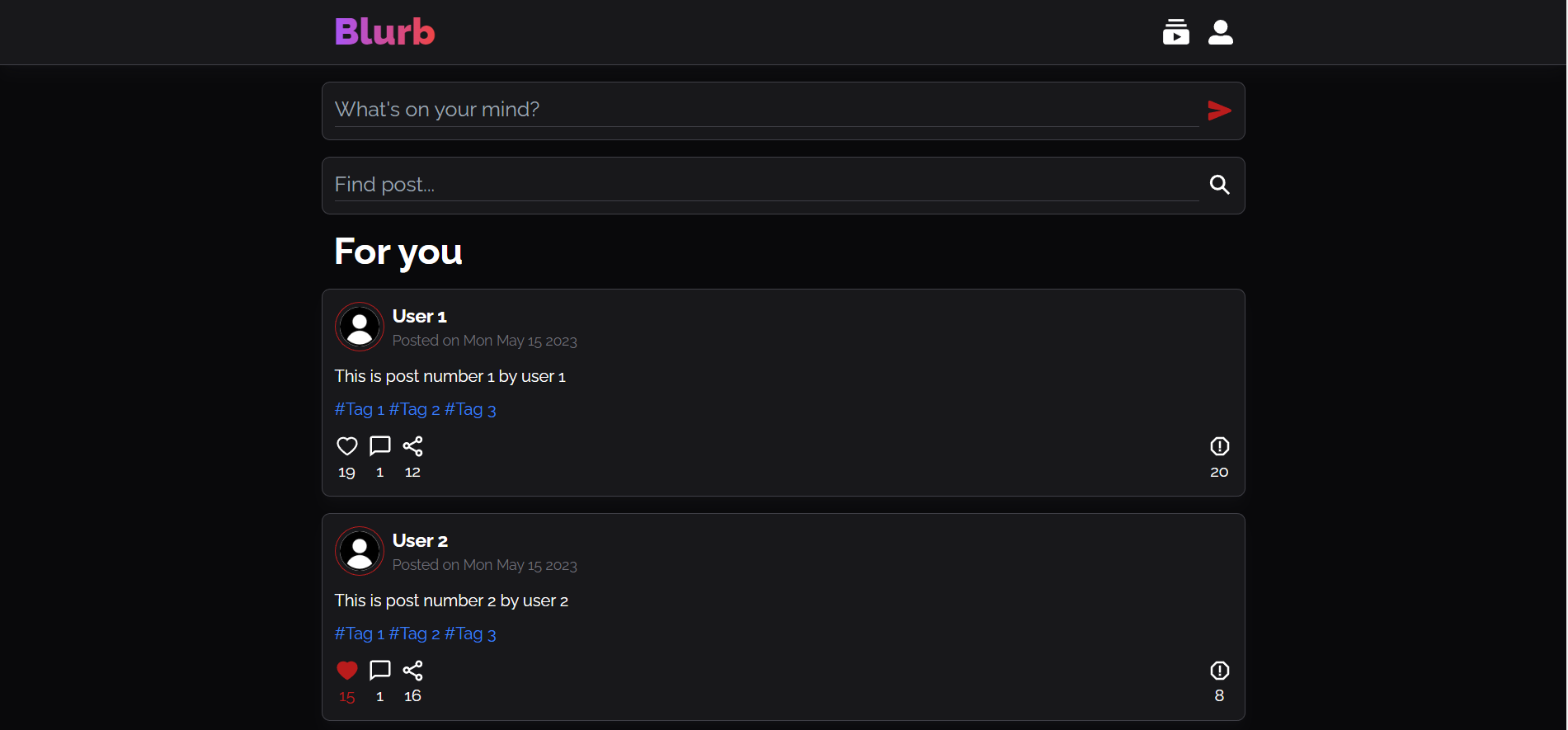


Рисунок 8.3 – Страница ленты новостей

Нажав на соответствующую кнопку пользователь может перейти на страницу своего профиля, представлено на рисунке 8.4. Здесь пользователь может изменить свой никнэйм, цвет оформления профиля и аватар пользователя. Тут пользователь может создать новую публикацию, а также просмотреть свои публикации.

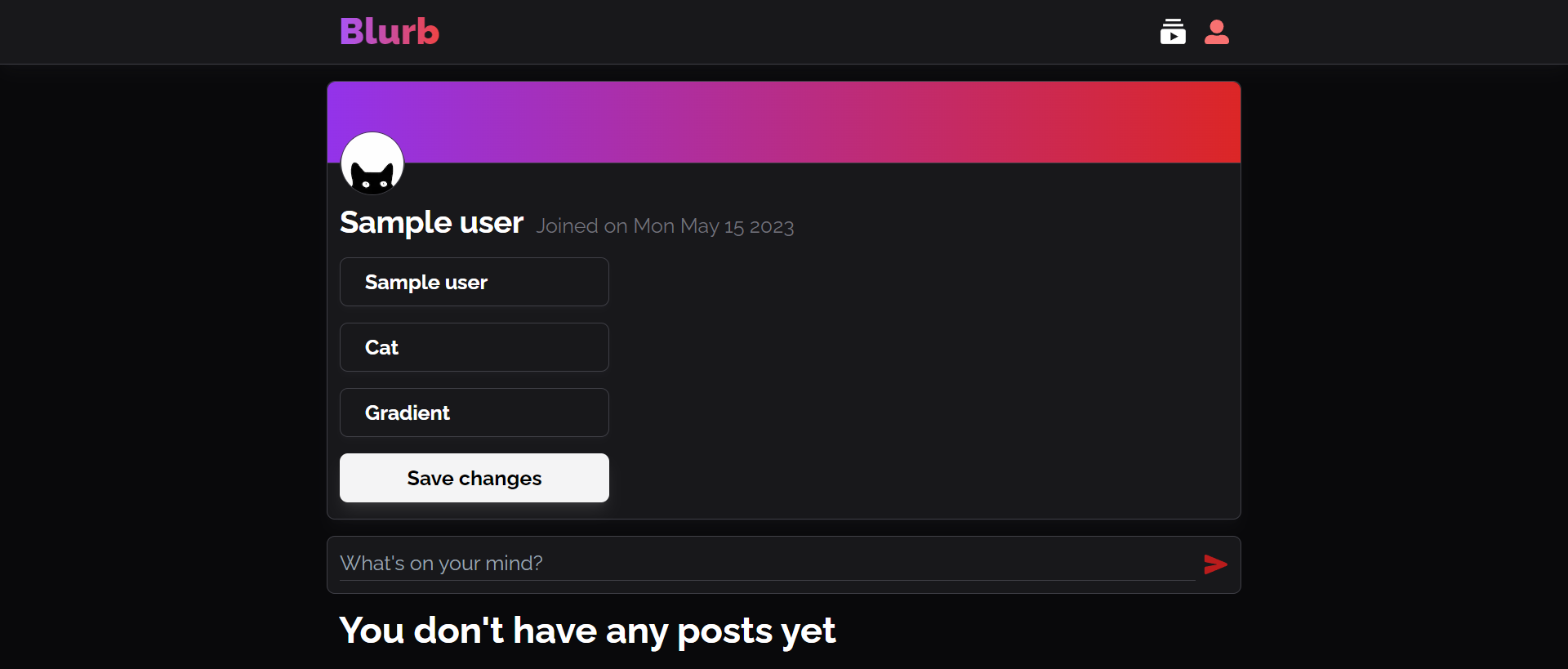


Рисунок 8.4 – Страница профиля

При нажатии на кнопку подписок пользователь попадает на соответствующую страницу, представлено на рисунке 8.5. Здесь пользователь может найти авторов и подписаться на них, просмотреть свои подписки.

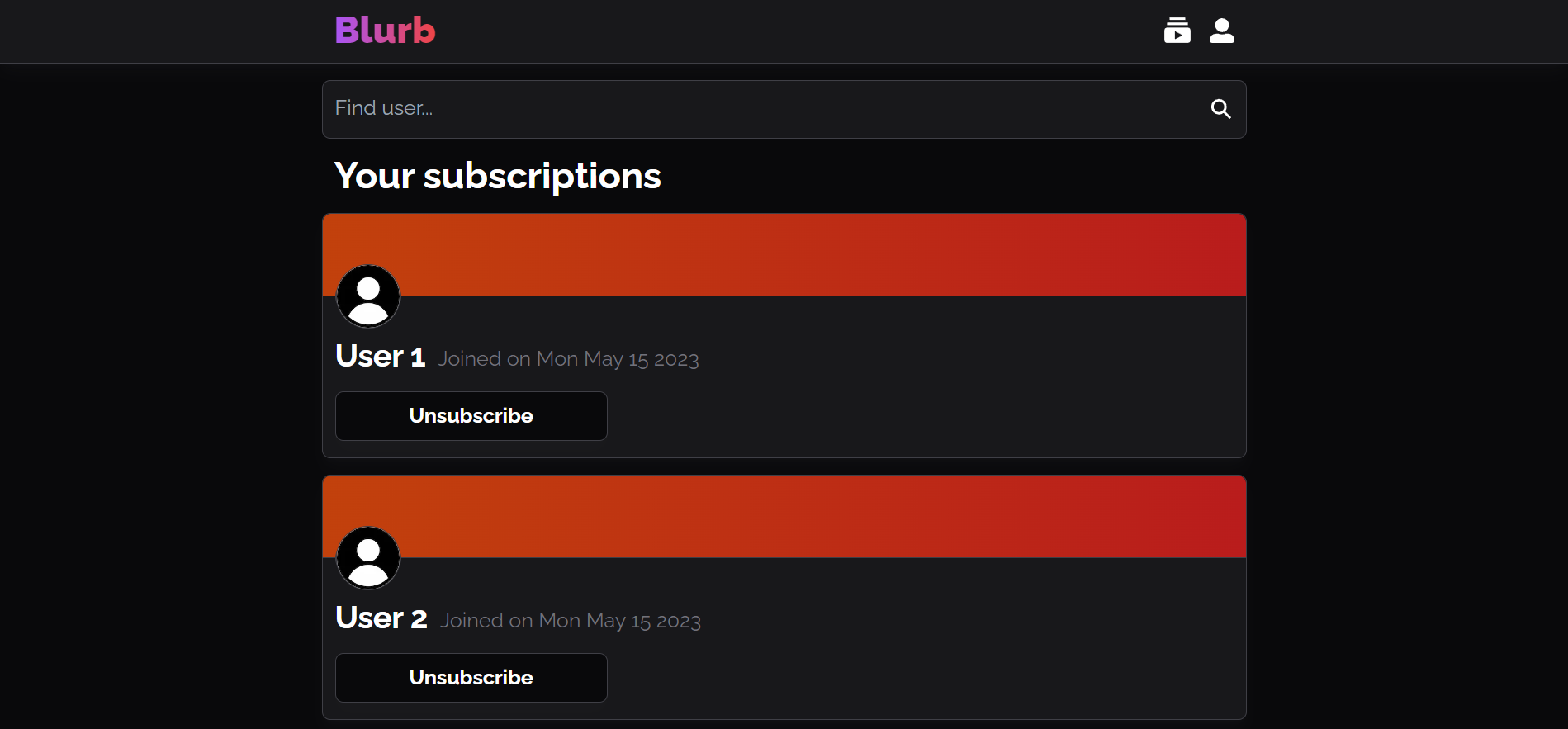


Рисунок 8.5 – Страница подписок

При просмотре публикаций пользователь может нажать на соответствующие кнопки чтобы поставить лайк, поделиться записью или пожаловаться. При нажатии на аватар или имя автора пользователь перейдет на его страницу, где он сможет подписаться, а при нажатии на кнопку комментариев пользователь попадет на страинцу публикации, где он сможет просмотреть комментарии других пользователей или оставить свой, представлено на рисуке 8.6.

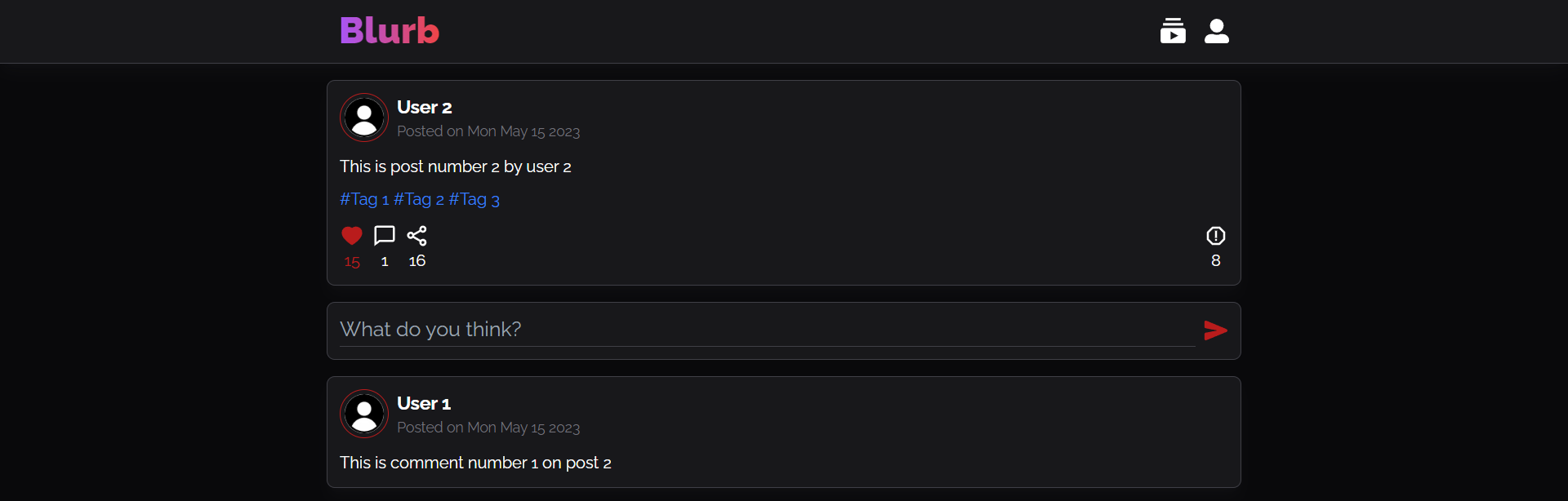


Рисунок 8.6 – Страница публикации

Если же пользователь является администратором или модератором, то на странице профиля появляется кнопка, позволяющая перейти в панель администрирования, представлено на рисунке 8.7. Здесь можно просмотреть статистику, просмотреть публикации с жалобами и предпринять соответствующие действия: убрать жалобы, удалить публикацию, удалить публикацию и заблокировать автора.

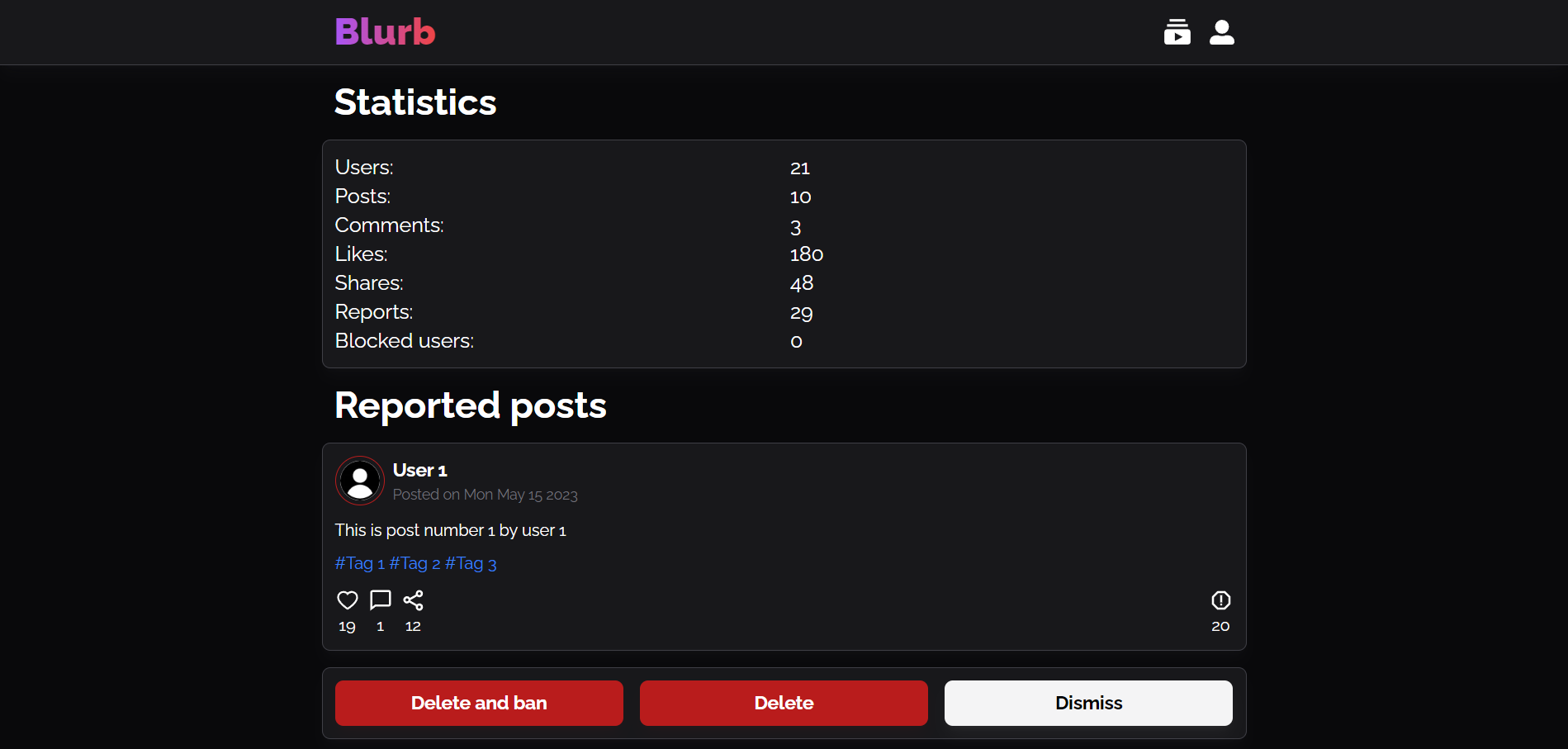


Рисунок 8.7 – Панель администрирования

**Заключение**

В ходе выполнения курсовой работы была выполнена поставленная цель по созданию базы данных и приложения для социальной сети. Удалось обеспечить высокую безопасность и гибкость базы данных. Благодаря набору разработанных процедур и функций удалось обеспечить сохранность данных от утечек и нежелательного воздействия, при этом не жертвуя ужобством разработки и использования. Благодаря использованию индексов удалось достичь высокой производительности базы данных.

При выполнени проекта были выполнены следующие задачи: проектирование инфраструктуры базы данных, разработка всех необходимых объектов и приложенния, реализация функциональных требований, тестирование производительности.

При разработке проекта выполнены следующие функции приложения:

˗ разработка приложения для взаимодействия с БД;

˗ регистрация и авторизация пользователей;

˗ обеспечение социальной функциональности;

˗ анализ работы сервиса

˗ мониторинг сети администратором.

В результате выполнения курсового проекта были получены навыки по взаимодействию с базой данных PostgreSQL, проведено сравнение решений SQL и NoSQL, работа с импортом и экспортом в JSON, а также разработки веб-приложения.

Список использованной литературы

1. PostgreSQL Documentation [Электронный ресурс] – Режим доступа: https://www.postgresql.org/docs/current/ - Дата доступа: 28.03.2023

2. Entity Framework Documentation Hub [Электронный ресурс] – Режим доступа: https://learn.microsoft.com/en-us/ef/ - Дата доступа: 25.04.2023

3. Руководство по ASP.NET Core 7 [Электронный ресурс] – Режим доступа: https://metanit.com/sharp/aspnet6/ - Дата доступа: 27.04.2023

4. Next.JS Docs [Электронный ресурс] – Режим доступа: https://nextjs.org/docs - Дата доступа: 28.04.2023

Приложение А (Создание таблиц и индексов)

CREATE TABLE "Users" (

"Id" integer GENERATED ALWAYS AS IDENTITY,

"Email" text UNIQUE NOT NULL,

"PasswordHash" text NOT NULL,

"Username" text NOT NULL,

"Avatar" text NOT NULL,

"ProfileColor" text NOT NULL,

"RegistrationDate" timestamp with time zone NOT NULL,

"Status" text NOT NULL,

"Role" text NOT NULL,

CONSTRAINT "PK\_Users" PRIMARY KEY ("Id")

);

CREATE TABLE "Posts" (

"Id" integer GENERATED ALWAYS AS IDENTITY,

"Text" text NOT NULL,

"CreationTime" timestamp with time zone NOT NULL,

"UserId" integer NULL,

CONSTRAINT "PK\_Posts" PRIMARY KEY ("Id"),

CONSTRAINT "FK\_Posts\_Users\_UserId"

FOREIGN KEY ("UserId") REFERENCES "Users" ("Id") ON DELETE CASCADE

);

CREATE TABLE "Subscriptions" (

"SubscriberId" integer NOT NULL,

"PublisherId" integer NOT NULL,

CONSTRAINT "PK\_Subscriptions" PRIMARY KEY ("SubscriberId", "PublisherId"),

CONSTRAINT "FK\_Subscriptions\_Users\_PublisherId"

FOREIGN KEY ("PublisherId") REFERENCES "Users" ("Id") ON DELETE CASCADE,

CONSTRAINT "FK\_Subscriptions\_Users\_SubscriberId"

FOREIGN KEY ("SubscriberId") REFERENCES "Users" ("Id") ON DELETE CASCADE

);

CREATE TABLE "Comments" (

"Id" integer GENERATED ALWAYS AS IDENTITY,

"PostId" integer NOT NULL,

"UserId" integer NULL,

"Text" text NOT NULL,

"CreationTime" timestamp with time zone NOT NULL,

CONSTRAINT "PK\_Comments" PRIMARY KEY ("Id"),

CONSTRAINT "FK\_Comments\_Posts\_PostId"

FOREIGN KEY ("PostId") REFERENCES "Posts" ("Id") ON DELETE CASCADE,

CONSTRAINT "FK\_Comments\_Users\_UserId"

FOREIGN KEY ("UserId") REFERENCES "Users" ("Id") ON DELETE CASCADE

);

CREATE TABLE "Likes" (

"UserId" integer NOT NULL,

"PostId" integer NOT NULL,

CONSTRAINT "PK\_Likes" PRIMARY KEY ("UserId", "PostId"),

CONSTRAINT "FK\_Likes\_Posts\_PostId"

FOREIGN KEY ("PostId") REFERENCES "Posts" ("Id") ON DELETE CASCADE,

CONSTRAINT "FK\_Likes\_Users\_UserId"

FOREIGN KEY ("UserId") REFERENCES "Users" ("Id") ON DELETE CASCADE

);

CREATE TABLE "Reports" (

"UserId" integer NOT NULL,

"PostId" integer NOT NULL,

CONSTRAINT "PK\_Reports" PRIMARY KEY ("UserId", "PostId"),

CONSTRAINT "FK\_Reports\_Posts\_PostId"

FOREIGN KEY ("PostId") REFERENCES "Posts" ("Id") ON DELETE CASCADE,

CONSTRAINT "FK\_Reports\_Users\_UserId"

FOREIGN KEY ("UserId") REFERENCES "Users" ("Id") ON DELETE CASCADE

);

CREATE TABLE "Shares" (

"UserId" integer NOT NULL,

"PostId" integer NOT NULL,

CONSTRAINT "PK\_Shares" PRIMARY KEY ("UserId", "PostId"),

CONSTRAINT "FK\_Shares\_Posts\_PostId"

FOREIGN KEY ("PostId") REFERENCES "Posts" ("Id") ON DELETE CASCADE,

CONSTRAINT "FK\_Shares\_Users\_UserId"

FOREIGN KEY ("UserId") REFERENCES "Users" ("Id") ON DELETE CASCADE

);

CREATE TABLE "Tags" (

"Id" integer GENERATED ALWAYS AS IDENTITY,

"Text" text NOT NULL,

"PostId" integer NULL,

CONSTRAINT "PK\_Tags" PRIMARY KEY ("Id"),

CONSTRAINT "FK\_Tags\_Posts\_PostId"

FOREIGN KEY ("PostId") REFERENCES "Posts" ("Id") ON DELETE CASCADE

);

CREATE INDEX "IX\_Users\_RegistrationDate" ON "Users" ("RegistrationDate");

CREATE INDEX "IX\_Posts\_CreationTime" ON "Posts" ("CreationTime");

CREATE INDEX "IX\_Posts\_UserId" ON "Posts" ("UserId");

CREATE INDEX "IX\_Comments\_CreationTime" ON "Comments" ("CreationTime");

CREATE INDEX "IX\_Comments\_UserId" ON "Comments" ("UserId");

CREATE INDEX "IX\_Comments\_PostId" ON "Comments" ("PostId");

CREATE INDEX "IX\_Likes\_UserId" ON "Likes" ("UserId");

CREATE INDEX "IX\_Likes\_PostId" ON "Likes" ("PostId");

CREATE INDEX "IX\_Reports\_UserId" ON "Reports" ("UserId");

CREATE INDEX "IX\_Reports\_PostId" ON "Reports" ("PostId");

CREATE INDEX "IX\_Shares\_UserId" ON "Shares" ("UserId");

CREATE INDEX "IX\_Shares\_PostId" ON "Shares" ("PostId");

CREATE INDEX "IX\_Subscriptions\_SubscriberId" ON "Subscriptions" ("SubscriberId");

CREATE INDEX "IX\_Subscriptions\_PublisherId" ON "Subscriptions" ("PublisherId");

CREATE INDEX "IX\_Tags\_PostId" ON "Tags" ("PostId");

CREATE INDEX "IX\_Tags\_Text" ON "Tags" ("Text");

Приложение Б (Создание триггеров и инфраструктуры для них)

CREATE TABLE "UserUpdateLogs" (

"Id" integer GENERATED ALWAYS AS IDENTITY,

"UserId" integer NOT NULL,

"UpdatedColumns" text[] NOT NULL,

"OldValues" text[] NOT NULL,

"NewValues" text[] NOT NULL,

"UpdateTimestamp" timestamp with time zone NOT NULL,

CONSTRAINT "PK\_UserUpdateLogs" PRIMARY KEY ("Id")

);

CREATE OR REPLACE FUNCTION log\_user\_update()

RETURNS TRIGGER AS $$

BEGIN

IF OLD."Email" = NEW."Email" AND OLD."RegistrationDate" = NEW."RegistrationDate" AND OLD."PasswordHash" = NEW."PasswordHash"

THEN

INSERT INTO "UserUpdateLogs" ("UserId", "UpdatedColumns", "OldValues", "NewValues", "UpdateTimestamp")

VALUES (

OLD."Id",

ARRAY(SELECT column\_name FROM information\_schema.columns

WHERE table\_name = 'Users' AND column\_name NOT IN ('Id', 'Email', 'RegistrationDate', 'PasswordHash')),

ARRAY[OLD."Username", OLD."Avatar", OLD."ProfileColor", OLD."Status", OLD."Role"],

ARRAY[NEW."Username", NEW."Avatar", NEW."ProfileColor", NEW."Status", NEW."Role"],

NOW()

);

END IF;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER user\_update\_trigger

AFTER UPDATE ON "Users"

FOR EACH ROW

EXECUTE FUNCTION log\_user\_update();

CREATE TABLE "CommentUpdateLogs" (

"Id" integer GENERATED ALWAYS AS IDENTITY,

"CommentId" integer NOT NULL,

"OldText" text NOT NULL,

"NewText" text NOT NULL,

"UpdateTimestamp" timestamp with time zone NOT NULL,

CONSTRAINT "PK\_CommentUpdateLogs" PRIMARY KEY ("Id")

);

CREATE OR REPLACE FUNCTION log\_comment\_update()

RETURNS TRIGGER AS $$

BEGIN

IF OLD."Text" IS DISTINCT FROM NEW."Text"

THEN

INSERT INTO "CommentUpdateLogs" ("CommentId", "OldText", "NewText", "UpdateTimestamp")

VALUES (

OLD."Id",

OLD."Text",

NEW."Text",

NOW()

);

END IF;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER comment\_update\_trigger

AFTER UPDATE ON "Comments"

FOR EACH ROW

EXECUTE FUNCTION log\_comment\_update();

CREATE TABLE "PostUpdateLogs" (

"Id" integer GENERATED ALWAYS AS IDENTITY,

"PostId" integer NOT NULL,

"OldText" text NOT NULL,

"NewText" text NOT NULL,

"UpdateTimestamp" timestamp with time zone NOT NULL,

CONSTRAINT "PK\_PostUpdateLogs" PRIMARY KEY ("Id")

);

CREATE OR REPLACE FUNCTION log\_post\_update()

RETURNS TRIGGER AS $$

BEGIN

IF OLD."Text" IS DISTINCT FROM NEW."Text"

THEN

INSERT INTO "PostUpdateLogs" ("PostId", "OldText", "NewText", "UpdateTimestamp")

VALUES (

OLD."Id",

OLD."Text",

NEW."Text",

NOW()

);

END IF;

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER post\_update\_trigger

AFTER UPDATE ON "Posts"

FOR EACH ROW

EXECUTE FUNCTION log\_post\_update();

Приложение В (Создание процедур и функций)

--Процедуры

CREATE OR REPLACE PROCEDURE add\_user(email text, password\_hash text, username text)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

INSERT INTO "Users" ("Email", "PasswordHash", "Username", "Avatar", "ProfileColor", "RegistrationDate", "Status", "Role")

VALUES (email, password\_hash, username, 'Basic', 'Red', NOW(), 'ACTIVE', 'USER');

END;

$$;

CREATE OR REPLACE PROCEDURE update\_user(id integer, profile\_color text DEFAULT NULL, avatar text DEFAULT NULL, username text DEFAULT NULL)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

UPDATE "Users"

SET "ProfileColor" = COALESCE(profile\_color, "ProfileColor"),

"Avatar" = COALESCE(avatar, "Avatar"),

"Username" = COALESCE(username, "Username")

WHERE "Id" = id;

END;

$$;

CREATE OR REPLACE PROCEDURE delete\_user(id integer)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

DELETE FROM "Users"

WHERE "Id" = id;

END;

$$;

CREATE OR REPLACE PROCEDURE update\_user\_status(id integer, status text)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

UPDATE "Users"

SET "Status" = status

WHERE "Id" = id;

END;

$$;

CREATE OR REPLACE PROCEDURE update\_user\_role(id integer, role text)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

UPDATE "Users"

SET "Role" = role

WHERE "Id" = id;

END;

$$;

CREATE OR REPLACE PROCEDURE update\_user\_password\_hash(id integer, password\_hash text)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

UPDATE "Users"

SET "PasswordHash" = password\_hash

WHERE "Id" = id;

END;

$$;

CREATE OR REPLACE PROCEDURE update\_user\_email(id integer, email text)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

UPDATE "Users"

SET "Email" = email

WHERE "Id" = id;

END;

$$;

CREATE OR REPLACE PROCEDURE add\_post(userid integer, text text)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

INSERT INTO "Posts" ("Text", "CreationTime", "UserId")

VALUES (text, NOW(), userid);

END;

$$;

CREATE OR REPLACE PROCEDURE update\_post(id integer, text text)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

UPDATE "Posts"

SET "Text" = text

WHERE "Id" = id;

END;

$$;

CREATE OR REPLACE PROCEDURE delete\_post(id integer)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

DELETE FROM "Posts"

WHERE "Id" = id;

END;

$$;

CREATE OR REPLACE PROCEDURE add\_comment(postid integer, userid integer, text text)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

INSERT INTO "Comments" ("PostId", "UserId", "Text", "CreationTime")

VALUES (postid, userid, text, NOW());

END;

$$;

CREATE OR REPLACE PROCEDURE update\_comment(id integer, text text)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

UPDATE "Comments"

SET "Text" = text

WHERE "Id" = id;

END;

$$;

CREATE OR REPLACE PROCEDURE delete\_comment(id integer)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

DELETE FROM "Comments"

WHERE "Id" = id;

END;

$$;

CREATE OR REPLACE PROCEDURE add\_tag(postid integer, text text)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

INSERT INTO "Tags" ("Text", "PostId")

VALUES (text, postid);

END;

$$;

CREATE OR REPLACE PROCEDURE delete\_tag(postid integer, tag text)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

DELETE FROM "Tags"

WHERE "Text" = tag AND "PostId" = postid;

END;

$$;

CREATE OR REPLACE PROCEDURE add\_tags(postid integer, tags text[])

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

INSERT INTO "Tags" ("Text", "PostId")

SELECT unnest(tags), postid;

END;

$$;

CREATE OR REPLACE PROCEDURE delete\_tags(id integer)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

DELETE FROM "Tags"

WHERE "Id" = id;

END;

$$;

CREATE OR REPLACE PROCEDURE add\_like(userid integer, postid integer)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

INSERT INTO "Likes" ("UserId", "PostId")

VALUES (userid, postid);

END;

$$;

CREATE OR REPLACE PROCEDURE add\_likes(userids integer[], postid integer)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

INSERT INTO "Likes" ("UserId", "PostId")

SELECT unnest(userids), postid;

END;

$$;

CREATE OR REPLACE PROCEDURE delete\_like(userid integer, postid integer)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

DELETE FROM "Likes"

WHERE "UserId" = userid AND "PostId" = postid;

END;

$$;

CREATE OR REPLACE PROCEDURE delete\_likes(postid integer)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

DELETE FROM "Likes"

WHERE "PostId" = postid;

END;

$$;

CREATE OR REPLACE PROCEDURE add\_share(userid integer, postid integer)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

INSERT INTO "Shares" ("UserId", "PostId")

VALUES (userid, postid);

END;

$$;

CREATE OR REPLACE PROCEDURE add\_shares(userids integer[], postid integer)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

INSERT INTO "Shares" ("UserId", "PostId")

SELECT unnest(userids), postid;

END;

$$;

CREATE OR REPLACE PROCEDURE delete\_share(userid integer, postid integer)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

DELETE FROM "Shares"

WHERE "UserId" = userid AND "PostId" = postid;

END;

$$;

CREATE OR REPLACE PROCEDURE delete\_shares(postid integer)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

DELETE FROM "Shares"

WHERE "PostId" = postid;

END;

$$;

CREATE OR REPLACE PROCEDURE add\_report(userid integer, postid integer)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

INSERT INTO "Reports" ("UserId", "PostId")

VALUES (userid, postid);

END;

$$;

CREATE OR REPLACE PROCEDURE add\_reports(userids integer[], postid integer)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

INSERT INTO "Reports" ("UserId", "PostId")

SELECT unnest(userids), postid;

END;

$$;

CREATE OR REPLACE PROCEDURE delete\_report(userid integer, postid integer)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

DELETE FROM "Reports"

WHERE "UserId" = userid AND "PostId" = postid;

END;

$$;

CREATE OR REPLACE PROCEDURE delete\_reports(postid integer)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

DELETE FROM "Reports"

WHERE "PostId" = postid;

END;

$$;

CREATE OR REPLACE PROCEDURE add\_subscription(subscriberid integer, publisherid integer)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

INSERT INTO "Subscriptions" ("SubscriberId", "PublisherId")

VALUES (subscriberid, publisherid);

END;

$$;

CREATE OR REPLACE PROCEDURE delete\_subscription(subscriberid integer, publisherid integer)

LANGUAGE plpgsql

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

DELETE FROM "Subscriptions"

WHERE "SubscriberId" = subscriberid AND "PublisherId" = publisherid;

END;

$$;

CREATE OR REPLACE PROCEDURE export\_posts\_json(path\_to\_file text)

LANGUAGE plpgsql

AS $$

BEGIN

EXECUTE format('COPY (SELECT row\_to\_json(t) FROM (SELECT \* FROM "Posts") t) TO %L', path\_to\_file);

END;

$$;

CREATE OR REPLACE PROCEDURE import\_posts\_json(path\_to\_file text)

LANGUAGE plpgsql

AS $$

BEGIN

CREATE TEMP TABLE temp (data jsonb);

EXECUTE format('COPY temp (data) FROM %L', path\_to\_file);

INSERT INTO "Posts" ("Text", "CreationTime", "UserId")

SELECT data->>'Text', (data->>'CreationTime')::timestamp with time zone, (data->>'UserId')::integer

FROM temp;

DROP TABLE temp;

END;

$$;

REVOKE EXECUTE ON PROCEDURE

add\_user, update\_user, update\_user\_status, update\_user\_role, update\_user\_password\_hash, update\_user\_email, delete\_user,

add\_post, update\_post, delete\_post,

add\_comment, update\_comment, delete\_comment,

add\_tag, add\_tags, delete\_tag, delete\_tags,

add\_like, add\_likes, delete\_like, delete\_likes,

add\_report, add\_reports, delete\_report, delete\_reports,

add\_share, add\_shares, delete\_share, delete\_shares,

add\_subscription, delete\_subscription,

export\_posts\_json, import\_posts\_json

FROM PUBLIC;

--Функции

CREATE OR REPLACE FUNCTION get\_users\_summary(id integer DEFAULT NULL)

RETURNS SETOF "UsersSummary"

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

IF id IS NULL THEN

RETURN QUERY SELECT \* FROM "UsersSummary";

ELSE

RETURN QUERY SELECT \* FROM "UsersSummary" WHERE "Id" = id;

END IF;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION get\_users\_summary\_by\_registration\_date(start\_date timestamp with time zone, end\_date timestamp with time zone)

RETURNS SETOF "UsersSummary"

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

RETURN QUERY SELECT \* FROM "UsersSummary" WHERE "RegistrationDate" BETWEEN start\_date AND end\_date;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION get\_posts\_summary(id integer DEFAULT NULL)

RETURNS SETOF "PostsSummary"

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

IF id IS NULL THEN

RETURN QUERY SELECT \* FROM "PostsSummary";

ELSE

RETURN QUERY SELECT \* FROM "PostsSummary" WHERE "Id" = id;

END IF;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION get\_posts\_summary\_by\_creation\_date(start\_date timestamp with time zone, end\_date timestamp with time zone)

RETURNS SETOF "PostsSummary"

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

RETURN QUERY SELECT \* FROM "PostsSummary" WHERE "CreationTime" BETWEEN start\_date AND end\_date;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION get\_trending\_posts()

RETURNS SETOF "PostsSummary"

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

RETURN QUERY SELECT \* FROM "PostsSummary" WHERE "CreationTime" > NOW() - INTERVAL '1 week' ORDER BY "LikeCount" DESC LIMIT 50;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION get\_reported\_posts()

RETURNS SETOF "PostsSummary"

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

RETURN QUERY SELECT \* FROM "PostsSummary" WHERE "ReportCount" > 0 ORDER BY "ReportCount" DESC;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION get\_comments\_on\_post(post\_id integer)

RETURNS SETOF "Comments"

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

RETURN QUERY

SELECT \* FROM "Comments" WHERE "PostId" = post\_id;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION get\_comments\_from\_user(user\_id integer)

RETURNS SETOF "Comments"

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

RETURN QUERY

SELECT \* FROM "Comments" WHERE "UserId" = user\_id;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION get\_subscriptions\_for\_publisher(publisher\_id integer)

RETURNS SETOF "Subscriptions"

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

RETURN QUERY

SELECT \* FROM "Subscriptions" WHERE "PublisherId" = publisher\_id;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION get\_subscriptions\_of\_user(subscriber\_id integer)

RETURNS SETOF "Subscriptions"

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

RETURN QUERY

SELECT \* FROM "Subscriptions" WHERE "SubscriberId" = subscriber\_id;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION get\_post\_likes(post\_id integer)

RETURNS SETOF "Likes"

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

RETURN QUERY

SELECT \* FROM "Likes" WHERE "PostId" = post\_id;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION get\_user\_likes(user\_id integer)

RETURNS SETOF "Likes"

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

RETURN QUERY

SELECT \* FROM "Likes" WHERE "UserId" = user\_id;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION get\_post\_reports(post\_id integer)

RETURNS SETOF "Reports"

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

RETURN QUERY

SELECT \* FROM "Reports" WHERE "PostId" = post\_id;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION get\_user\_reports(user\_id integer)

RETURNS SETOF "Reports"

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

RETURN QUERY

SELECT \* FROM "Reports" WHERE "UserId" = user\_id;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION get\_post\_shares(post\_id integer)

RETURNS SETOF "Shares"

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

RETURN QUERY

SELECT \* FROM "Shares" WHERE "PostId" = post\_id;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION get\_user\_shares(user\_id integer)

RETURNS SETOF "Shares"

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

RETURN QUERY

SELECT \* FROM "Shares" WHERE "UserId" = user\_id;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION get\_post\_tags(post\_id integer)

RETURNS SETOF "Tags"

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

RETURN QUERY

SELECT \* FROM "Tags" WHERE "PostId" = post\_id;

END;

$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION get\_popular\_tags()

RETURNS TABLE (Id integer, Text text, PostId integer, Usage integer)

SECURITY DEFINER

SET search\_path = public, pg\_temp

AS $$

BEGIN

RETURN QUERY

SELECT "Tags"."Text", COUNT("Tags"."PostId") AS "Usage"

FROM "Tags"

GROUP BY "Tags"."Text"

ORDER BY COUNT("Tags"."PostId") DESC;

END;

$$ LANGUAGE plpgsql;

REVOKE EXECUTE ON FUNCTION

get\_users\_summary, get\_users\_summary\_by\_registration\_date,

get\_posts\_summary, get\_trending\_posts, get\_reported\_posts,

get\_comments\_on\_post, get\_comments\_from\_user,

get\_subscriptions\_for\_publisher, get\_subscriptions\_of\_user,

get\_post\_likes, get\_user\_likes,

get\_post\_reports, get\_user\_reports,

get\_post\_shares, get\_user\_shares,

get\_post\_tags, get\_popular\_tags

FROM PUBLIC;