Министерство образования Республики Беларусь
Учреждение образования «Витебский государственный
университет имени П.М. Машерова»
Кафедра прикладного и системного программирования

А.И. Никитин

СОВРЕМЕННЫЕ ПОСТРЕЛЯЦИОННЫЕ БАЗЫ ДАННЫХ

Методические рекомендации к выполнению лабораторных работ

Витебск
ВГУ имени П.М. Машерова
2024

УДК 004.65(076.5)
ББК 16.3я73
Н62

Печатается по решению научно-методического совета учреждения образования «Витебский государственный университет имени П.М. Машерова». Протокол № 3 от 29.02.2024.

Автор: доцент кафедры прикладного и системного программирования ВГУ имени П.М. Машерова, кандидат физико-математических наук **А.И. Никитин**

Р е ц е н з е н т ы :
заведующий кафедрой «Математика и информационные технологии» УО «ВГТУ», кандидат физико-математических наук, доцент *Т.В. Никонова*;
доцент кафедры информационных технологий и управления бизнесом ВГУ имени П.М. Машерова, кандидат физико-математических наук, доцент *С.А. Прохожий*

Никитин, А.И.

Н62 Современные постреляционные базы данных : методические рекомендации к выполнению лабораторных работ / А.И. Никитин. – Витебск : ВГУ имени П.М. Машерова, 2024. – 33 с.

В методических рекомендациях показан ход выполнения лабораторных работ по различным темам, помогающим студентам освоить проектирование реляционных баз данных, а также детально изучается система управления базами данных PostgreSQL. Практически по всем темам приводятся различные примеры, демонстрирующие те или иные возможности представленных технологий.

Предназначается для студентов специальности 1-31 03 07-01 02 Прикладная информатика (программное обеспечение компьютерных систем).

УДК 004.65(076.5)
ББК 16.3я73

СОДЕРЖАНИЕ

Введение 4

Лабораторная работа № 1. Проектирование баз данных 5
1.1. Ход лабораторной работы 5
1.2. Самостоятельная работа 7

Лабораторная работа № 2. Проектирование базы данных в PostgreSQL 11
2.1. Работа с рабочей средой PostgreSQL 11
2.2. Создание базы данных в PostgreSQL 12
2.3. Самостоятельная работа 14

Лабораторная работа № 3. Выборка данных в PostgreSQL 14
3.1. Ход лабораторной работы 14
3.2. Самостоятельная работа «Запросы» 19

Лабораторная работа № 4. Транзакции 21
4.1. Ход лабораторной работы 21
4.2. Самостоятельная работа «Уровни изоляции транзакций» 24

Лабораторная работа № 5. Индексы 26
5.1. Ход лабораторной работы 26
5.2. Самостоятельная работа «Скорость запроса» 30
5.3. Самостоятельная работа «Индексы» 30

Список литературы 32

ВВЕДЕНИЕ

В настоящее время термин «база данных» известен многим людям, даже далеким от профессиональной разработки компьютерных программ. Базы данных стали очень широко распространенной технологией, что потребовало, в свою очередь, большего числа специалистов, способных проектировать их и обслуживать. В ходе эволюции теории и практики баз данных стандартом де-факто стала реляционная модель данных, а в рамках этой модели сформировался и специализированный язык программирования, позволяющий выполнять все необходимые операции с данными – Structured Query Language (SQL). Таким образом, важным компонентом квалификации специалиста в области баз данных является владение языком SQL.

По охвату тем в методических рекомендациях рассматриваются темы о проектировании реляционных баз данных, а также детально изучается система управления базами данных PostgreSQL. Практически по всем темам приводятся различные примеры, демонстрирующие те или иные возможности представленных технологий.

ЛАБОРАТОРНАЯ РАБОТА № 1.
ПРОЕКТИРОВАНИЕ БАЗ ДАННЫХ

1.1. Ход лабораторной работы

Проектирование базы данных – это процесс создания структурированного плана организации, хранения и управления данными для обеспечения целостности, согласованности и эффективности данных. Хорошо спроектированная база данных снижает избыточность данных, обеспечивает возможность повторного использования и упрощает управление данными. Проектирование хорошей базы данных предполагает использование лучших практик и методов, таких как моделирование данных, нормализация и моделирование отношений сущностей.

Моделирование данных создает графическое представление структуры базы данных, определяя сущности, атрибуты и связи для точного представления реальных сценариев. Модель данных служит основой для физического и логического проектирования базы данных. Обычно процесс включает в себя следующие этапы:

1. **Анализ требований**: выявление и сбор требований заинтересованных сторон и понимание целей и задач системы.
2. **Концептуальная модель данных**: модель высокого уровня, которая представляет основные сущности, атрибуты и отношения без рассмотрения деталей структуры базы данных. Эта независимая от технологий модель фокусируется на структуре хранимых данных.
3. **Логическая модель данных**: подробная модель, которая дополнительно расширяет концептуальную модель данных, определяя все необходимые сущности, атрибуты, отношения и ограничения в структурированном формате. Эта модель открывает путь к физическому проектированию базы данных.
4. **Реализация физической модели данных**. Используя логическую модель данных в качестве руководства, база данных создается и заполняется данными путем определения таблиц, индексов и других объектов базы данных.

Выполнив эти шаги, можно создать прочную основу для своей базы данных и гарантировать, что она точно отражает нужды и требования вашей организации.

Рассмотрим пример проектирование баз данных для создания интернет-магазина. Конкретизируем нашу предметную область.

Пусть имеет некий интернет-магазин, который продает некоторые товары, которые разбиты по категориям. Пользователи могут ложить товары в корзину и в последствии оформлять заказы. В корзину могут ложить товары как авторизированные пользователи, так и неавторизиро-

ванные. Заказ может быть доставлен различными способами доставки. Так же имеются несколько способов оплаты, в том числе – возможность оплатить заказ с помощью кошелька в интернет-магазине, который можно пополнять.

Для начала нужно выделить основные сущности, которые буду использовать в базе данных. Для нашей предметной области получим следующий список:

1. Пользователь
2. Продукт
3. Категория продукта.
4. Корзина
5. Заказ
6. Способ доставки
7. Способ оплаты

После определения основных сущностей нужно определить связи между ними:

1. В одной категории могут лежать несколько продуктов и каждый продукт привязан к категории. Будем считать, что один продукт не может лежать в нескольких категориях.

2. Каждый продукт могут ложить разные пользователи в корзину и причем несколько раз, т.е. один и тот же продукт может участвовать в разных заказах.

3. Продукт, находящий в корзине может как быть привязанным к заказу, так и не быть привязанным к заказу.

4. Доставка, как и оплата привязывается к заказу.

5. Заказ может быть оплачен несколькими способами, например, часть может быть оплачена, например, банковской картой, а вторая часть – из кошелька интернет-магазина.

После установления связей между сущностями, добавляются все необходимые поля, а также устанавливаются базовые ограничения. На основе всех приведенных данных можем получить следующую схему базы данных (Рисунок 1).

Полученная схема базы данных представляет концептуальную модель и может использоваться в дальнейшем для создания физической модели.

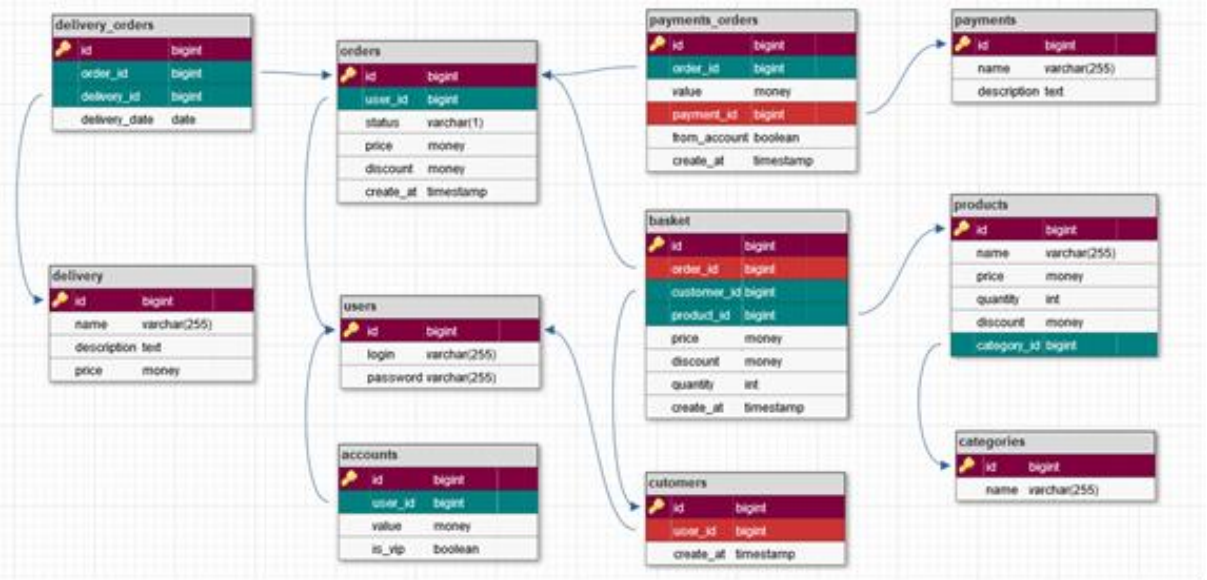

Рисунок 1. Схема базы данных

1.2. Самостоятельная работа

Спроектировать инфологическую модель базы данных и создать ER-диаграмму, соответствующую спроектированной модели. Обосновать целесообразность спроектированной модели.

Для проектирования базы данных можно использовать один из следующих онлайн-сервисов:

1. https://dbdesigner.net/
2. https://dbdsgnr.appspot.com/
3. https://www.draw.io/
4. https://erdplus.com/#/

Варианты:

1. **Доставка еды.** Сервис должен предоставлять возможность заказа еды пользователями. Торговые компании могут регистрироваться в системе и выставлять свои товары. Пользователь собирает нужные ему товары в корзину и оформляет заказ на доставку. Также должна быть возможность написания отзывов компаниям.
2. **Выгул животных**. Сервис предоставляет возможность заказ услуги на выгул и уход за животными. 2 типа пользователей: клиент – может регистрировать своих животных в системе и выставлять заявки на выгул или уход; петситтер – может выполнять заявки оставленные клиентами. Также должна быть возможность написания отзывов петситтерам.
3. **Клининг**. Сервис должен предоставлять возможность заказа различных услуг по уборке офисов и квартир. Пользователь может выбирать различные услуги и формировать цены в зависимости от необходимой площади уборки. После завершения услуги пользователь должен делать

оценку уборки. Также должна быть возможность написания отзывов по отдельным услугам.

4. **Такси**. Сервис предоставляет возможность заказа такси. 2 типа пользователей: клиент – может размещать заказы на поездки с выбранным типом комфорта, цена формируется автоматические исходя из километража (можно брать из API карт); водитель – может выполнять заказы на поездку. Также должна быть возможность написания отзыва по окончанию поездки.

5. **Аукцион**. Сервис предоставляет возможность размещения слотов для аукциона сроком на 12, 24 или 48 часов. Пользователи могут выставлять минимальную цену и цену выкупа, а также шаг ставки в %. Слоты должны располагаться по категориям. Пользователи должны иметь возможность отслеживать выбранные ими слоты. Должна вестись статистика по самым успешным пользователям и по самым дорогим лотам.

6. **Каршеринг**. Сервис предоставляет возможность аренды машин на время. Пользователь должен иметь возможность найти ближайшую машину по карте и арендовать ее на определенный срок. Компания может управлять машинами автопарка. Цена формируется исходя из марки машины и срока аренды. Работа происходит по предоплате. Также должна быть возможность написания отзыва о конкретной машине.

7. **Турниры по киберспорту.** Сервис предоставляет возможность по организации различными клиентами турниров по киберспорту по системе double elimination. Организатор турнира имеет возможность создать турнир, а команды подать заявку на участие в нем. Сетка должна генерироваться автоматически исходя из количества команд. Организатор имеет право вносить туда результаты. По результатам соревнования должны выводится итоговые места.

8. **Сервис по бронированию мест в отелях**. Сервис предоставляет возможность бронировать места в отелях. Менеджер отеля может зарегистрировать свой отель и внести в базу номера и их описания. Пользователи могут бронировать места на определенный срок с постоплатой. Также менеджер также имеет возможность бронировать места по своему усмотрению. Также должна быть возможность написания отзывов по отдельным номерам.

9. **Планирование питания.** Сервис должен отслеживать питание пользователей по калорийности, белкам, жирам, углеводам и давать рекомендации (самые простые). Сервис имеет базу продуктов и блюд и информацию о их калорийности и т.д. Пользователь может добавлять свои блюда в систему через заявку администратору. Также пользователь вносит информацию о том, какие он блюда употреблял. Система выдает статистику по питанию и рекомендации к его изменению.

10. **Бронирование столиков в ресторане.** Сервис предоставляет возможность бронировать столиков в ресторане. Менеджер отеля может

зарегистрировать свой ресторан и внести в базу столы и схему их расположения. Пользователи могут бронировать столики на определенную дату и время. Также менеджер также имеет возможность бронировать столики по своему усмотрению. Также должна быть возможность написания отзывов по отдельным ресторанам.

11. **Обмен книгами.** Сервис предоставляет возможность по размещению печатных версий книг с целью обмена или продажи. Пользователи могут размещать книги, покупать их, а также менять на те, которые разместили. Под каждую книгу можно оставлять рецензии и цитаты.

12. **Hh.ru.** Сервис по поиску работы и предоставлении вакансий. Компании могут размещать свои вакансии, пользователи могут на них откликаться. Пользователи могут размещать свои резюме, а компании могут на них откликаться. Также должна быть возможность написания отзывов о компаниях.

13. **Сливки.** Сервис предоставляет различные скидочные купоны. Компания может регистрироваться в системе и размещать ограниченное количество купонов на определенный период на различные предоставляемые услуги. Пользователи могут покупать купоны и оставлять отзывы о компаниях.

14. **Ставки на события.** Система может создавать различные события (это может быть все что угодно, например исходы на спорт, на результат отчисления студента и прочее) и делать различные исходы на их результат. Остальные пользователи могут делать ставки на различные исходы и получают за угаданные результаты балы. Также пользователи могут оставлять комментарии под каждым событием. Должна вестись статистика по самым популярным событиям (максимальное количество ставок), а также топ самых крутых предсказателей.

15. **Запись к врачам.** Сервис предоставляет возможность записи в различные учреждения на различные услуги. Менеджер поликлиники может размещать талоны по различным услугам для пользователей, а пользователи соответственно записываться к врачу по талону. Также должна быть возможность написания отзывов о поликлинике.

16. **Рецепты по ингредиентам.** Сервис должен позволять по определенным продуктам получать рецепты. Пользователи могут размещать рецепты с указанием состава продуктов, которые имеются в системе. Также они могут выбрать определенные продукты с учетом веса и сервис должен показывать, что из этого можно приготовить. Под каждым рецептом можно оставлять отзывы. Продукты в систему заносятся администратором либо через заявку пользователем.

17. **Поиск учебного заведения.** Сервис предоставляет возможность поиск специальностей высших учебных заведений по сданным ЦТ. Менеджер учреждения образования может регистрировать его в системе (через

заявку) и размещать специальности с описанием и необходимым ЦТ. Абитуриенты могут искать нужные им специальности по ЦТ, которым они сдавали. Для каждого учебного заведения можно писать отзывы.

18. **Инстаграм для музыкантов.** Сервис предоставляет возможность размещения музыки различными музыкантами и исполнителями. Пользователи могут загружать различную музыку, а другие пользователи могут комментировать эти произведения и выставлять лайки. Также пользователи могут вести составлять список подписок на различных музыкантов, записи которых будут выводится у них на стене.

19. **Сервис по отзывам на компании.** Сервис предоставляет возможность комментирования и оставления отзывов по компаниям по категориям различных услуг. Менеджер компании может подавать заявку на регистрацию компании в системе. После принятия заявки он может добавлять информацию о ней. Пользователи могут оставлять отзывы о компании, а менеджер компании имеет возможность отвечать на эти отзывы. Должна быть возможность оценки отзывов и ответов менеджера (аналог лайков).

20. **Кинотеатр.** Сервис предоставляет возможность бронирования билетов на сеансы в кинотеатре. Администратор может добавлять различные фильмы в систему и сеансы на эти фильмы. Каждый сеанс может проходить в различных залах кинотеатра. Пользователи могут бронировать места на сеанс. Предусмотреть возможность оставления отзывов о фильмах.

21. **Организация онлайн-курсов.** Сервис предоставляет возможность записи на курсы и посещения занятий. Пользователи могут записываться на имеющиеся курсы. Преподаватели, которые ведут курсы могу составлять расписание и выставлять оценки пользователи при посещении занятий.

22. **Туристические маршруты.** Сервис предоставляет возможность составления туристических маршрутов и организации туров по ним. Имеется база различных мест, которые можно посетить. Гид может составить произвольный туристический маршрут из имеющих мест и добавить его в сервис. Пользователи могут записываться на туристические маршруты для участия в них. Должна быть система отзывов по различным местам.

23. **Навигатор общественного транспорта.** Сервис предоставляет возможность просмотра расписания общественного транспорта. Имеется набор остановок в некотором городе. Администратор может добавлять общественный транспорт с указанием его маршрута, направления и расписания по будням выходным. Пользователь имеет возможность просматривать информацию по отдельным маршрутам, а также смотреть информацию о расписании транспорта на конкретные остановки (должна выводится информация об ближайшем транспорте в интервале часа). Предусмотреть возможность просмотра информации через API карт.

ЛАБОРАТОРНАЯ РАБОТА № 2.
ПРОЕКТИРОВАНИЕ БАЗЫ ДАННЫХ В POSTGRESQL

2.1. Работа с рабочей средой PostgreSQL

Начать нужно с выбора того дистрибутива СУБД, который будет установлен. Можно выбрать оригинальный вариант PostgreSQL или тот, который предлагается компанией Postgres Professional. Он называется Postgres Pro и содержит не только все функции и модули, входящие в состав стандартного дистрибутива, но и дополнительные разработки, выполненные в компании Postgres Professional. Для изучения основ языка SQL эти дистрибутивы подходят в равной степени. Однако документация на русском языке включена только в состав Postgres Pro.

После того как определен конкретный дистрибутив СУБД, необходимо выбрать операционную систему. PostgreSQL поддерживает множество систем, в том числе различные версии Linux, а также Windows.

Устанавливать рекомендуется последнюю стабильную версию СУБД.

Если необходимо использоваться оригинальный дистрибутив PostgreSQL, то найти инструкции по его установке в различных операционных системах можно по адресу https://www.postgresql.org/download/.

После установки как PostgreSQL, так и Postgres Pro, в среде Windows придется предпринять дополнительные меры, чтобы использование русского алфавита в интерактивном терминале psql не вызывало проблем. Установив тот или иной дистрибутив PostgreSQL, нужно научиться запускать сервер баз данных, потому что иначе невозможно работать с данными. Как это сделать, подробно описано в документации в разделе 18.3 «Запуск сервера баз данных».

Для доступа к серверу баз данных в комплект PostgreSQL входит интерактивный терминал psql. Для его запуска нужно ввести команду

Psql

При запуске утилиты psql в среде Windows возможно некорректное отображение букв русского алфавита. Для устранения этого потребуется в свойствах окна, в котором выполняется psql, изменить шрифт на Lucida Console и с помощью команды chcp сменить текущую кодовую страницу на CP1251:

chcp 1251

В среде утилиты psql можно вводить не только команды языка SQL, но и различные сервисные команды, поддерживаемые самой утилитой.

Для получения краткой справки по всем сервисным командам нужно ввести

\?

Многие такие команды начинаются с символов «\d». Например, для того чтобы просмотреть список всех таблиц и представлений (views), созданных в той базе данных, к которой вы сейчас подключены, введите команду

\dt

Если же вас интересует определение (попросту говоря, структура) какой-либо конкретной таблицы базы данных, например, students, нужно ввести команду

\d students

Для получения списка всех SQL-команд нужно выполнить команду

\h

Для вывода описания конкретной SQL-команды, например, CREATE TABLE, нужно сделать так:

\h CREATE TABLE

2.2. Создание базы данных в PostgreSQL.

База данных создаётся SQL-командой CREATE DATABASE:

CREATE DATABASE имя;

где имя подчиняется правилам именования идентификаторов SQL. Текущий пользователь автоматически назначается владельцем. Владелец может удалить свою базу, что также приведёт к удалению всех её объектов, в том числе, имеющих других владельцев.

Для создания таблиц в языке SQL служит команда **CREATE TABLE**. Ее полный синтаксис представлен в документации на PostgreSQL, а упрощенный синтаксис таков:

CREATE TABLE имя-таблицы
(
имя-поля тип-данных [ограничения-целостности],
имя-поля тип-данных [ограничения-целостности],
...
имя-поля тип-данных [ограничения-целостности],
[ограничение-целостности],
[первичный-ключ],
[внешний-ключ]
);

В квадратных скобках показаны необязательные элементы команды. После команды

нужно поставить символ «;».

Пример создания таблицы **products** может быть представлен следующий образом:

```
CREATE TABLE public.products (
    "id" serial NOT NULL,
    name character varying(255),
    price money,
    quantity integer,
    discount money,
    category_id bigint NOT NULL
);
```

Чтобы удалить таблицу из базы данных можно использовать следующую команду:

```
DROP TABLE имя-таблицы;
```

Отметим, что в каждой таблицы будет присутствовать первичный ключи. Как правило им в каждой таблицы является поле “ID”. Несмотря на то, что он не всегда является обязательным и в некоторых случаях можно обойтись без него, все равно рекомендуется использовать его как первичный ключ, так как на этапе разработки серверных приложений множество решений (библиотек, пакетов и т.д.) требуют его в качестве первичного ключа. Создать первичный ключ можно следующим образом:

```
ALTER TABLE ONLY products
    ADD CONSTRAINT products_pkey PRIMARY KEY ("id");
```

Кроме того в нашей таблице также имеется внешний ключ **category_id,** который связывает наш продукт с определенной категорией. Его можно создать следующим образом:

```
ALTER TABLE ONLY products
    ADD CONSTRAINT products_categories FOREIGN KEY (category_id) REFERENCES categories("id") NOT VALID;
```

Как правило, все внешние ключи прописываются только после описания всех таблиц, чтобы исключить вероятность ошибки в порядке описания таблиц в базе данных.

Вставка данных в созданную таблицу можно совершить, используя следующую команду:

```
INSERT INTO products VALUES ( 'Бананы', 10.2, 5, 0.0, 1);
```

Если есть необходимость вставить сразу несколько записей, то это можно сделать следующим образом:

```
INSERT INTO products VALUES
    ( 'Бананы', 10.2, 5, 0.0, 1),
    ( 'Хлеб', 1.2, 7, 0.0, 2),
    ( 'Ананас', 15.2, 1, 1.0, 1),
    ( 'Яйца', 3.2, 20, 0.2, 2);
```

2.3. Самостоятельная работа

Создать скрипты для инициализации и удаления базы данных из Лабораторной работы №1. Протестировать работу скриптов.

Создать скрипт для заполнения базы данных тестовой информацией в объёме, достаточном, чтобы продемонстрировать все особенности спроектированной модели (не менее 5 осмысленных записей для каждой таблицы).

ЛАБОРАТОРНАЯ РАБОТА № 3.
ВЫБОРКА ДАННЫХ В POSTGRESQL

3.1. Ход лабораторной работы

Для выборки информации из таблиц базы данных служит команда SELECT. Ее синтаксис, упрощенный до предела, таков:

```
SELECT имя-атрибута, имя-атрибута, ...
FROM имя-таблицы;
```

Часто бывает так, что требуется вывести значения из всех столбцов таблицы. В таком случае можно не перечислять имена атрибутов, а просто ввести символ «*». Запрос, который выводит всю информацию из таблицы **products** будет выглядеть следующим образом:

```
SELECT * FROM products;
```

При выполнении простой выборки из таблицы СУБД не гарантирует никакого конкретного порядка вывода строк. Если же вы хотите каким-то образом упорядочить расположение выводимых строк, то необходимо предпринять дополнительные меры, а именно: использовать предложение ORDER BY команды SELECT. Пример:

```
SELECT name, price
FROM products
ORDER BY price DESC;
```

Для того чтобы ограничить число строк, включаемых в результирующую выборку, служит предложение LIMIT.

```
SELECT name, price
FROM products
ORDER BY price DESC;
LIMIT 3;
```

А как вывести не первые три продукта, а те, которые занимают места с четвертого по шестое? Алгоритм будет почти таким же, но он будет дополнен еще одним шагом: нужно пропустить три первые строки, прежде чем начать вывод. Для пропуска строк служит предложение OFFSET.

```
SELECT name, price
FROM products
ORDER BY price DESC;
LIMIT 3
OFFSET 3;
```

Далеко не всегда требуется выбирать все строки из таблицы. Множество выбираемых строк можно ограничить с помощью предложения WHERE команды SELECT. Выберем продукты, у которых цена находится в пределах от 10 до 20 включительно.

```
SELECT name, price
FROM products
WHERE price >= 10 AND price <= 20;
```

Условие выбора строк может быть составным. В данном случае мы скомбинировали два ограничения с помощью логической операции AND (т. е. «И»).

Выбрать все продукты, которые начинаются на букву «Б». Для этого можно использовать оператор поиска шаблонов LIKE:

```
SELECT *
FROM products
WHERE name LIKE 'Б%';
```

В тех случаях, когда информации, содержащейся в одной таблице, недостаточно для получения требуемого результата, используют соединение (**JOIN**) таблиц. Приведем пример, когда нужно вывести все продукты, которые находятся в категории «Фрукты»:

```
SELECT p.name, p.price, c.name
FROM products AS p JOIN categories AS c ON p.category_id = c.id
WHERE c.name = 'Фрукты';
```

Данная команда иллюстрирует соединение двух таблиц на основе равенства значений атрибутов. В этой команде в предложении FROM указаны две таблицы – **products** и **categories**, причем каждая из них получила еще и псевдоним с помощью ключевого слова AS (заметим, что оно не является обязательным). Конечно, псевдонимы могут состоять не только из одной буквы, как в нашем примере. Псевдонимы удобны в тех случаях, когда в соединяемых таблицах есть одноименные атрибуты. В таких случаях в списке атрибутов, следующих за ключевым словом SELECT, необходимо указывать либо имя таблицы, из которой выбирается значение этого атрибута, либо ее псевдоним, но псевдоним может быть коротким, что удобнее при написании команды. Псевдоним и атрибут соединяются символом «.». Псевдонимы используются и в предложениях WHERE, GROUP BY, ORDER BY, HAVING, т. е. во всех частях команды SELECT.

Кроме простого JOIN существуют еще виды соединения таблиц: LEFT JOIN, RIGHT JOIN, FULL JOIN, INNER JOIN.

INNER JOIN. С помощью этого запроса вы возвратите все записи из таблиц table_1 и table_2, которые связаны с помощью первичного (primary) и внешнего (foreign) ключей, а также отвечающие условию WHERE для таблицы table_1. Если в какой-нибудь из вышеописанных таблиц отсутствует запись, которая соответствует соседней, эта пара не будет включена в общую выдачу. Таким образом, мы получим лишь те записи, которые существует как в первой, так и во второй таблицах. По сути, выборка осуществляется по наличию связи (ключу), то есть выдаются лишь записи, связанные между собой. Если у вас есть «одинокие» записи (записи без пары), то они выданы не будут.

LEFT JOIN. С помощью этого запроса вы вернёте все данные из «левой» таблицы даже в том случае, если не будет найдено соответствий в «правой» таблице. Подразумевается, что «левая» таблица в запросе находится левее знака равно, а «правая», соответственно, правее (стандартная логика правой и левой руки). Говоря иначе, когда мы присоединяем «правую» таблицу к «левой», происходит выборка всех записей согласно условиям WHERE для «левой» таблицы. Если в «правой» таблице у нас отсутствуют соответствующие записи по ключам, они вернутся как NULL. В результате главной выступает именно «левая» таблица, и именно относительно неё осуществляется выдача. При этом в условии ON «левая» таблица прописывается первой по порядку (table_1), а «правая» – второй (table_2).

RIGHT JOIN. Используя этот запрос, вы вернёте все данные из «правой» таблицы даже в том случае, если не будут найдены соответствия в «левой» таблице. То есть всё происходит по аналогии с LEFT JOIN, однако NULL вернётся для полей «левой» таблицы. Иными словами, главной

выступает именно правая «таблица» и выдача осуществляется относительно неё. Также обратите внимание на WHERE, т. к. условие выборки теперь затрагивает «правую» таблицу.

FULL JOIN является объединением LEFT JOIN и RIGHT JOIN.

В команде SELECT предусмотрены средства для выполнения операций с выборками,

как с множествами, а именно:

– UNION для вычисления объединения множеств строк из двух выборок;

– INTERSECT для вычисления пересечения множеств строк из двух выборок;

– EXCEPT для вычисления разности множеств строк из двух выборок.

Запросы должны возвращать одинаковое число столбцов, типы данных у столбцов также должны совпадать.

Например, выберем список продуктов из категории «Фрукты» и «Овощи».

```
SELECT p.name, p.price, c.name
FROM products AS p JOIN categories AS c ON p.category_id = c.id
WHERE c.name = 'Фрукты'
UNION
SELECT p.name, p.price
FROM products AS p JOIN categories AS c ON p.category_id = c.id
WHERE c.name = 'Овощи';
```

Среди множества функций, имеющихся в PostgreSQL, важное место занимают агрегатные функции. Для работы с агрегатными функциями используется команда GROUP BY. GROUP BY собирает в одну строку все выбранные строки, выдающие одинаковые значения для выражений группировки. В качестве выражения внутри элемента группирования может выступать имя входного столбца, либо имя или порядковый номер выходного столбца (из списка элементов SELECT), либо произвольное значение, вычисляемое по значениям входных столбцов.

GROUP BY элемент_группирования

Для группировки могу использоваться следующие функции:

Функция	Типы аргумента	Тип результата	Описание
avg(выражение)	smallint, int, bigint, real, double precision, numeric или interval	Numeric для любых целочисленных аргументов, double precision для аргументов с плавающей точкой, в противном случае тип данных аргумента	арифметическое среднее для всех входных значений, отличных от NULL

count(*)		bigint	количество входных строк
count(выражение)	any	bigint	количество входных строк, для которых значение выражения не равно NULL
max(выражение)	любой числовой, строковый, сетевой тип или тип даты/времени, либо массив этих типов	тот же, что и тип аргумента	максимальное значение выражения среди всех входных данных, отличных от NULL
min(выражение)	любой числовой, строковый, сетевой тип или тип даты/времени, либо массив этих типов	тот же, что и тип аргумента	минимальное значение выражения среди всех входных данных, отличных от NULL
sum(выражение)	smallint, int, bigint, real, double precision, numeric, interval или money	bigint для аргументов smallint или int, numeric для аргументов bigint, и тип аргумента в остальных случаях	сумма значений выражения по всем входным данным, отличным от NULL

Выведем, например, количество товаров в каждой категории:

```
SELECT c.id, c.name, count(*)
FROM products AS p RIGHT JOIN categories AS c ON p.category_id =
c.id
GROUP BY c.id;
```

При выполнении выборок можно с помощью условий, заданных в предложении WHERE, сузить множество выбираемых строк. Аналогичная возможность существует и при выполнении группировок: можно включить в результирующее множество не все строки, а лишь те, которые удовлетворяют некоторому условию. Это условие можно задать в предложении HAVING. Важно помнить, что предложение WHERE работает с отдельными строками еще до выполнения группировки с помощью GROUP BY, а предложение HAVING – уже после выполнения группировки.

В качестве примера выведем категории, где больше 10 товаров:

```
SELECT c.id, c.name, count(*)
FROM products AS p RIGHT JOIN categories AS c ON p.category_id =
c.id
GROUP BY c.id
HAVING count(*) > 10;
```

Теперь рассмотрим команду UPDATE, предназначенной для обновления данных в таблицах. Ее упрощенный синтаксис таков:

```
UPDATE имя-таблицы
        SET   имя-атрибута1 = значение-атрибута1,
              имя-атрибута2 = значение-атрибута2, ...
        WHERE условие;
```

Условие, указываемое в команде, должно ограничить диапазон обновляемых строк. Если это условие не задать, то будут обновлены все строки в таблице. Если же вам требуется обновить лишь часть из них, то не забывайте указывать условие отбора строк для обновления.

Увеличим, например, все цена на 10% из категории 1.

```
UPDATE products
        SET price = price * 1.1
        WHERE category_id = 1;
```

Для удаления строк из таблицы используется команда DELETE, которая похожа на команду SELECT:

```
DELETE FROM имя-таблицы
      WHERE условие;
```

Удалите какую-нибудь одну строку из таблицы products:

```
DELETE FROM products WHERE id = 2;
```

3.2. Самостоятельная работа «Запросы»

Для представленной схемы базы данных интернет-магазина в Лабораторной №1 напишите набор запросов.

1. Вывести количество товаров для каждой категории.
2. Вывести список пользователей, которые не оплатили хотя бы 1 заказ полностью.
3. Удалить товары, которые ни разу не были куплены (отсутствуют информация в таблице basket).
4. Вывести заказы, которые оплачены только частично.
5. Вывести для каждого покупателя количество его заказов по статусам.
6. Вывести средний чек для выполненных заказов (status="F").

7. Вывести топ-10 самых продаваемых товаров по суммарной прибыли (заказы для этих товаров должны быть полностью оплачены).

8. Вывести список товаров, которые лежат пока только в корзине и не привязаны к заказу, и их количества не хватает на складе для продажи (считается, что товар списывается со склада только тогда, когда заказ будет доставлен).

9. Вывести список пользователей, которые "бросили" свои корзины (не оформили заказ) за последние 30 дней.

10. Добавить скидку 10% на все товары, которые покупались (статус заказов "P" или "F") не более 10 раз.

11. Вывести количество заказов оплаченных полностью с внутреннего счета.

12. Сделать скидку 50% (без доставки, только на товары) на новые заказы (статус "N") для VIP-пользователей.

13. Вывести самую популярную доставку и самый популярный способ оплаты (результат из 2 записей)

14. Удалить пустые категории.

15. Вывести список пользователей, которые были оплачены полностью не более чем через час, с момента добавления первого товара в корзину.

16. Вернуть все деньги пользователей на внутренний счет для заказов, которые были оплачены (как внутренним счетом, так и некоторым способом оплаты) и были отменены (status = "C").

Указания:

1. Поля, выделенные бордовым цветом, являются первичными ключами;
2. Поля, выделенные зеленым цветом, являются внешними ключами с атрибутом ***NOT NULL***;
3. Поля, выделенные оранжевым цветом, являются внешними ключами, которые могут иметь значение ***NULL***;
4. Таблица **accounts** (внутренний счет в магазине, используемый для оплаты): *value* - количество денег на счете, *is_vip* - VIP-счет;
5. Таблица **customers** (список покупателей, которые по-умолчанию имеют статус неавторизированного пользователя (*user_id=**NULL***));
6. Таблица **basket** (содержит записи о положенных в корзину товарах. Если *order_id=**NULL***, значит товары еще не прикреплены ни к одному заказу, множество ({*order_id, customer_id, product_id*}) не содержим одинаковых значений): *price* - цена одного товара с учетом скидки, *discount* - величина скидки, *quantity* - количество товара в корзине;
7. Таблица **orders** (список заказов): *price* - общая сумма заказа (товары+доставка) с учетом скидки, *discount* - величина скидки, *status* - статус

заказа ("N" - новый, "D" - доставлен, "P" - оплачен, "F" - выполнен, "C" - отменен), статус заказа никак не привязан к данным в таблицах ***payments_orders*** и ***delivery_orders*** (например, статус "D" не гарантирует наличие записи в таблице ***delivery_orders***);

8. Таблица **payments_orders** (оплаты заказов, для одного заказа может быть несколько оплат): *value* - величина оплаты, если *payment_id* ***IS NOT NULL*** и *from_account* = ***FALSE***, значит оплата из справочника ***payments***, если *payment_id* ***IS NULL*** и *from_account* = ***TRUE***, значит оплата произведена со внутреннего счета ***accounts***;

9. Таблица **products** (список товаров): *price* - цена товара с учетом скидки, *discount* - величина скидки, *quantity* - количество товара на складе;

10. Все данные в таблицах считать корректными и не имеющие никаких противоречий.

ЛАБОРАТОРНАЯ РАБОТА № 4. ТРАНЗАКЦИИ

4.1. Ход лабораторной работы

Транзакция – это совокупность операций над базой данных, которые вместе образуют логически целостную процедуру, и могут быть либо выполнены все вместе, либо не будет выполнена ни одна из них. В простейшем случае транзакция состоит из одной операции.

Транзакции являются одним из средств обеспечения согласованности (непротиворечивости) базы данных, наряду с ограничениями целостности (constraints), накладываемыми на таблицы. Транзакция переводит базу данных из одного согласованного состояния в другое согласованное состояние.

В качестве примера примера транзакции можно привести процесс создания заказа в интернет-магазине. Этот процесс состоит из нескольких этапов:

1. Создание записи в таблице orders.
2. Обновление записей в таблице basket (привязка к заказу)
3. Обновление остатков продукции в таблице products.
4. Добавление записей в таблицы delivery_orders и payment_orders.

Если эти операции будут выполнены не все, например, не спишутся остатки товаров при заказе, то другие покупатели будут видеть не корректные значения количества продуктов, т.е. возможно увидят товар, который уже купили и его в наличии.

Транзакция может иметь два исхода: первый – изменения данных, произведенные в ходе ее выполнения, успешно зафиксированы в базе

данных, а второй исход таков – транзакция отменяется, и отменяются все изменения, выполненные в ее рамках. Отмена транзакции называется откатом (rollback).

Сложные информационные системы, как правило, предполагают одновременную работу многих пользователей с базой данных, поэтому современные СУБД предлагают специальные механизмы для организации параллельного, т. е. одновременного, выполнения транзакций. Реализованы такие механизмы и в PostgreSQL.

Реализация транзакций в СУБД PostgreSQL основана на многоверсионной модели (Multiversion Concurrency Control, MVCC). Эта модель предполагает, что каждый SQLоператор видит так называемый снимок данных (snapshot), т. е. то согласованное состояние (версию) базы данных, которое она имела на определенный момент времени. При этом параллельно исполняемые транзакции, даже вносящие изменения в базу данных, не нарушают согласованности данных этого снимка. Такой результат в PostgreSQL достигается за счет того, что когда параллельные транзакции изменяют одни и те же строки таблиц, тогда создаются отдельные версии этих строк, доступные соответствующим транзакциям. Это позволяет ускорить работу с базой данных, однако требует больше дискового пространства и оперативной памяти. И еще одно важное следствие применения MVCC – операции чтения никогда не блокируются операциями записи, а операции записи никогда не блокируются операциями чтения.

Согласно теории баз данных транзакции должны обладать следующими свойствами:

1. **Атомарность** (atomicity). Это свойство означает, что либо транзакция будет зафиксирована в базе данных полностью, т. е. будут зафиксированы результаты выполнения всех ее операций, либо не будет зафиксирована ни одна операция транзакции.

2. **Согласованность** (consistency). Это свойство предписывает, чтобы в результате успешного выполнения транзакции база данных была переведена из одного согласованного состояния в другое согласованное состояние.

3. **Изолированность** (isolation). Во время выполнения транзакции другие транзакции должны оказывать по возможности минимальное влияние на нее.

4. **Долговечность** (durability). После успешной фиксации транзакции пользователь должен быть уверен, что данные надежно сохранены в базе данных и впоследствии могут быть извлечены из нее, независимо от последующих возможных сбоев в работе системы.

Для обозначения всех этих четырех свойств используется аббревиатура ACID.

При параллельном выполнении транзакций возможны следующие феномены:

1. **Потерянное обновление** (lost update). Когда разные транзакции одновременно изменяют одни и те же данные, то после фиксации изменений может оказаться, что одна транзакция перезаписала данные, обновленные и зафиксированные другой транзакцией.

2. **«Грязное» чтение** (dirty read). Транзакция читает данные, измененные параллельной транзакцией, которая еще не завершилась. Если эта параллельная транзакция в итоге будет отменена, тогда окажется, что первая транзакция прочитала данные, которых нет в системе.

3. **Неповторяющееся чтение** (non-repeatable read). При повторном чтении тех же самых данных в рамках одной транзакции оказывается, что другая транзакция успела изменить и зафиксировать эти данные. В результате тот же самый запрос выдает другой результат.

4. **Фантомное чтение** (phantom read). Транзакция повторно выбирает множество строк в соответствии с одним и тем же критерием. В интервале времени между выполнением этих выборок другая транзакция добавляет новые строки и успешно фиксирует изменения. В результате при выполнении повторной выборки в первой транзакции может быть получено другое множество строк.

5. **Аномалия сериализации** (serialization anomaly). Результат успешной фиксации группы транзакций, выполняющихся параллельно, не совпадает с результатом ни одного из возможных вариантов упорядочения этих транзакций, если бы они выполнялись последовательно.

Для конкретизации степени независимости параллельных транзакций вводится понятие уровня изоляции транзакций. Каждый уровень характеризуется перечнем тех феноменов, которые на данном уровне не допускаются.

Всего в стандарте SQL предусмотрено четыре уровня. Каждый более высокий уровень включает в себя все возможности предыдущего.

1. Read Uncommitted. Это самый низкий уровень изоляции. Согласно стандарту SQL на этом уровне допускается чтение «грязных» (незафиксированных) данных. Однако в PostgreSQL требования, предъявляемые к этому уровню, более строгие, чем в стандарте: чтение «грязных» данных на этом уровне не допускается.

2. Read Committed. Не допускается чтение «грязных» (незафиксированных) данных. Таким образом, в PostgreSQL уровень Read Uncommitted совпадает с уровнем Read Committed. Транзакция может видеть только те незафиксированные изменения данных, которые произведены в ходе выполнения ее самой.

3. Repeatable Read. Не допускается чтение «грязных» (незафиксированных) данных и неповторяющееся чтение. В PostgreSQL на этом уровне не допускается также фантомное чтение. Таким образом, реализация этого

уровня является более строгой, чем того требует стандарт SQL. Это не противоречит стандарту.

4. Serializable. Не допускается ни один из феноменов, перечисленных выше, в том числе и аномалии сериализации.

Конкретный уровень изоляции обеспечивает сама СУБД с помощью своих внутренних механизмов. Его достаточно указать в команде при старте транзакции. Однако программист может дополнительно использовать некоторые операторы и приемы программирования, например, устанавливать блокировки на уровне отдельных строк или всей таблицы.

По умолчанию PostgreSQL использует уровень изоляции Read Committed.

4.2. Самостоятельная работа «Уровни изоляции транзакций»

1. Транзакции, работающие на уровне изоляции Read Committed, видят только свои собственные обновления и обновления, зафиксированные параллельными транзакциями. При этом нужно учитывать, что иногда могут возникать ситуации, которые на первый взгляд кажутся парадоксальными, но на самом деле все происходит в строгом соответствии с этим принципом.

Воспользуемся таблицей products или ее копией. Предположим, что мы решили удалить из таблицы те продукты, цена которых менее 20. В таблице представлена один такой продукт «Хлеб», имеющий цену 10. Для выполнения удаления мы организовали транзакцию. Однако параллельная транзакция, которая, причем, началась раньше, успела обновить таблицу таким образом, что цена продукта «Хлеб» 30, а вот для продукта «Сок» она, напротив, уменьшилась до 15. Таким образом, в результате выполнения операций обновления в таблице по-прежнему присутствует строка, удовлетворяющая первоначальному условию, т. е. значение атрибута price у которой меньше 20.

Проверить, будет ли в результате выполнения двух транзакций удалена какая-либо строка из таблицы. Объяснить результат.

2. Когда говорят о таком феномене, как потерянное обновление, то зачастую в качестве примера приводится операция UPDATE, в которой значение какого-то атрибута изменяется с применением одного из действий арифметики. Например:

```
UPDATE products
SET price = price + 2
WHERE category_id = 1;
```

При выполнении двух и более подобных обновлений в рамках параллельных транзакций, использующих, например, уровень изоляции Read Committed, будут учтены все такие изменения. Очевидно, что потерянного обновления не происходит.

Предположим, что в одной транзакции будет просто присваиваться новое значение, например, так:

```
UPDATE products
SET price = 10
WHERE category_id = 1;
```

А в параллельной транзакции будет выполняться аналогичная команда:

```
UPDATE products
SET price = 12
WHERE category_id = 1;
```

Очевидно, что сохранится только одно из значений атрибута price. Можно ли говорить, что в такой ситуации имеет место потерянное обновление? Если оно имеет место, то что можно предпринять для его недопущения? Обоснуйте ваш ответ.

3. На уровне изоляции транзакций Read Committed имеет место такой феномен, как чтение фантомных строк. Такие строки могут появляться в выборке как в результате добавления новых строк параллельной транзакцией, так и вследствие изменения ею значений атрибутов, участвующих в формировании условия выборки. Рассмотрим пример, иллюстрирующий вторую из указанных причин. На первом терминале организуем транзакцию. Она будет иметь уровень изоляции Read Committed:

```
BEGIN;

SELECT *
FROM products
WHERE price > 60;
```

На втором терминале организуем транзакцию и обновим одну из строк таблицы таким образом, чтобы эта строка стала удовлетворять условию отбора строк, заданному в первой транзакции.

```
BEGIN;

UPDATE products
SET price = 61
WHERE name LIKE 'A%';
```

Сразу завершим вторую транзакцию, чтобы первая транзакция увидела эти изменения.

END;

На первом терминале повторим ту же самую выборку:

```
SELECT *
FROM products
WHERE price > 60;
```

Транзакция еще не завершилась, но она уже увидела новую строку, обновленную зафиксированной параллельной транзакцией. Теперь эта строка стала соответствовать условию выборки. Таким образом, не изменяя критерий выборки, мы получили другое множество строк.

Завершим теперь и первую транзакцию:

END;

Модифицируйте этот пример: вместо операции UPDATE используйте операцию INSERT.

4. В тексте лабораторной работы приведен механизм создания заказа. Используя механизм транзакций, напиши SQL-код, который создает заказ с учетом всех необходимых данных и операций, подобрав соответствующий уровень изоляции и объяснив его выбор.

ЛАБОРАТОРНАЯ РАБОТА № 5. ИНДЕКСЫ

5.1. Ход лабораторной работы

Строки в таблицах хранятся в неупорядоченном виде. При выполнении операций выборки, обновления и удаления СУБД должна отыскать нужные строки. Для ускорения этого поиска и создается индекс. В принципе он 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. 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. Все это определяет разработчик базы данных при создании индекса.

Для того чтобы увидеть индексы, созданные для данной таблицы, нужно воспользоваться командой утилиты psql:

\d имя-таблицы

Например,

\d products

...
Индексы:
"products_pkey" PRIMARY KEY, btree (id)
"products_id_key" UNIQUE CONSTRAINT, btree (id)
...

Каждый индекс, который был создан самой СУБД, имеет типовое имя, состоящее из следующих компонентов:

– имени таблицы и суффикса pkey – для первичного ключа;

– имени таблицы, имен столбцов, по которым создан индекс, и суффикса key – для уникального ключа.

В описании также присутствует список столбцов, по которым создан индекс, и тип индекса – в данном случае это btree, т.е. B-дерево. PostgreSQL может создавать индексы различных типов, но по умолчанию используется так называемое B-дерево. Такой индекс подходит для большинства типовых задач. В этой главе мы будем рассматривать только индексы на основе B-дерева.

Наличие индекса может ускорить выборку строк из таблицы, если он создан по столбцам, на основе значений которых и производится выборка. Поэтому, как правило, при разработке и эксплуатации баз данных не ограничиваются только индексами, которые автоматически создает СУБД, а создают дополнительные индексы с учетом наиболее часто выполняющихся выборок.

Для создания индексов предназначена команда

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

В этой команде имя индекса можно не указывать. В качестве примера давайте создадим индекс для таблицы products по столбцу name.

```
CREATE INDEX
ON products (name);
```

Посмотрим описание нового индекса:

```
\d products
...
Индексы:
...
"products_name_idx" btree (name)
...
```

Обратите внимание, что имя индекса, сформированное автоматически, включает имя таблицы, имя столбца и суффикс idx.

Прежде чем приступить к экспериментам с индексами, нужно включить в утилите psql секундомер с помощью следующей команды:

```
\timing on
```

Когда необходимость в использовании секундомера отпадет, для его отключения нужно будет сделать так:

```
\timing off
```

Теперь psql будет сообщать время, затраченное на выполнение всех команд.

Просмотреть список всех индексов в текущей базе данных можно командой

```
\di
```

Или

```
\di+
```

Для удаления индекса используется команда:

```
DROP INDEX имя-индекса;
```

Удалим созданный нами индекс для таблицы products:

```
DROP INDEX products_name_idx;
```

Когда индекс уже создан, о его поддержании в актуальном состоянии заботится СУБД.

Конечно, следует учитывать, что это требует от СУБД затрат ресурсов и времени. Индекс, созданный по столбцу, участвующему в соединении двух таблиц, может позволить ускорить процесс выборки записей из таблиц. При выборке записей в отсортированном порядке индекс также может помочь, если сортировка выполняется по тем столбцам, по которым индекс создан.

Индексы могут также использоваться для обеспечения уникальности значений атрибутов в строках таблицы. В таком случае создается уникальный индекс. Для его создания используется команда:

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

В команде создания индекса можно использовать не только имена столбцов, но также функции и скалярные выражения, построенные на основе столбцов таблицы. Например, если мы захотим запретить значения столбца name в таблице products, отличающиеся только регистром символов, то создадим такой индекс:

```
CREATE UNIQUE INDEX products_unique_name_key
ON products (lower( name) );
```

PostgreSQL поддерживает очень интересный тип индексов – частичные индексы. Такой индекс формируется не для всех строк таблицы, а лишь для их подмножества.

Это достигается с помощью использования условного выражения, называемого предикатом индекса. Предикат вводится с помощью предложения WHERE.

В качестве иллюстрации создадим частичный индекс для таблицы orders. Представим, что руководство компании интересуют заказы на сумму свыше 100 рублей. Такая выборка выполняется с помощью запроса

```
SELECT *
FROM orders
WHERE price > 100
ORDER BY create_at DESC;
```

Хотя сортировка строк производится по датам бронирования в убывающем порядке, т. е. от более поздних дат к более ранним, тем не менее, включать ключевое слово DESC в индексное выражение, когда индекс создается только по одному столбцу, нет необходимости. Это объясняется

тем, что PostgreSQL умеет совершать обход индекса как по возрастанию, так и по убыванию с одинаковой эффективностью.

Обратите внимание, что индексируемый столбец create_at не участвует в формировании предиката индекса – в предикате используется столбец price. Это вполне допустимая ситуация.

```
CREATE INDEX orders_create_at_part_key
ON bookings (create_at)
WHERE price > 100;
```

Частичные индексы выглядят очень привлекательно, но в большинстве случаев их преимущества по сравнению с обычными индексами будут минимальными. Однако размер частичного индекса будет меньше, чем размер обычного. Для получения заметного полезного эффекта от их применения необходимы опыт и понимание того, как работают индексы в PostgreSQL.

5.2. Самостоятельная работа «Скорость запроса»

Для Вашей представленной базы данных из Лабораторной работы № 1 сгенерировать тестовые данные. Разработать запрос, который будет делать выборку по крайней мере из 3 связных таблиц, использовать фильтрацию, сортировку и вывод конкретного диапазона записей (WHERE, ORDER BY, LIMIT). Запрос должен выполняться несколько секунд (для этого можно увеличить количество тестовых данных или сложность запроса). Добавить необходимые индексы к Вашему запросу, чтобы скорость запроса была увеличена в несколько раз (возможно предложить его оптимизацию). Вывести план запроса и объяснить, за счет чего была получена прибавка в скорости выполнения запроса.

5.3. Самостоятельная работа «Индексы»

1. Предположим, что для какой-то таблицы создан уникальный индекс по двум столбцам: column1 и column2. В таблице есть строка, у которой значение атрибута column1 равно ABC, а значение атрибута column2 – NULL. Мы решили добавить в таблицу еще одну строку с такими же значениями ключевых атрибутов, т. е. column1 – ABC, а column2 – NULL. Как вы думаете, будет ли операция вставки новой строки успешной или завершится с ошибкой? Объясните ваше решение

2. Для одной из таблиц создайте индекс по двум столбцам, причем по одному из них укажите убывающий порядок значений столбца, а по другому – возрастающий. Значения NULL у первого столбца должны распола-

гаться в начале, а у второго – в конце. Посмотрите полученный индекс с помощью команд psql

\d имя_таблицы
\di+ имя_индекса

Обратите внимание, что первая команда выведет не только имя индекса, но также и имена столбцов, по которым он создан, а вторая команда выведет размер индекса.

Подберите запросы, в которых созданный индекс предположительно должен использоваться, а также запросы, в которых он использоваться, по вашему мнению, не будет. Проверьте ваши гипотезы, выполнив запросы. Объясните полученные результаты.

3. В сложных базах данных целесообразно использование комбинаций индексов. Иногда бывают более полезны комбинированные индексы по нескольким столбцам, чем отдельные индексы по единичным столбцам. В реальных ситуациях часто приходится делать выбор, т. е. находить компромисс, между, например, созданием двух индексов по каждому из двух столбцов таблицы либо созданием одного индекса по двум столбцам этой таблицы, либо созданием всех трех индексов. Выбор зависит от того, запросы какого вида будут выполняться чаще всего. Предложите какую-нибудь таблицу в базе данных интернет-магазина и смоделируйте ситуации, в которых вы приняли бы одно из этих трех возможных решений. Воспользуйтесь документацией на PostgreSQL.

СПИСОК ЛИТЕРАТУРЫ

1. Молинаро, Э. SQL. Сборник рецептов. – Пер. с англ. – СПб: Символ-Плюс, 2009. – 672 с.

2. Ригс, С. Администрирование PostgreSQL 9. Книга рецептов / С. Ригс, Х. Кроссинг // пер. с англ. Самохвалова Е.В. – М.: ДМК Пресс, 2012. – 368 с.

3. Connolly, T.M. Database Systems. A Practical Approach to Design Implementation and Management – 6th Global Edition / T.M. Connolly, C.E. Begg. – London: Pearson, 2014. – 1440 p.

4. Schönig, H.-J. Mastering PostgreSQL 13. Build, administer, and maintain database applications efficiently with PostgreSQL 13. – Birmingham: Packt Publishing, 2020. – 476 p.