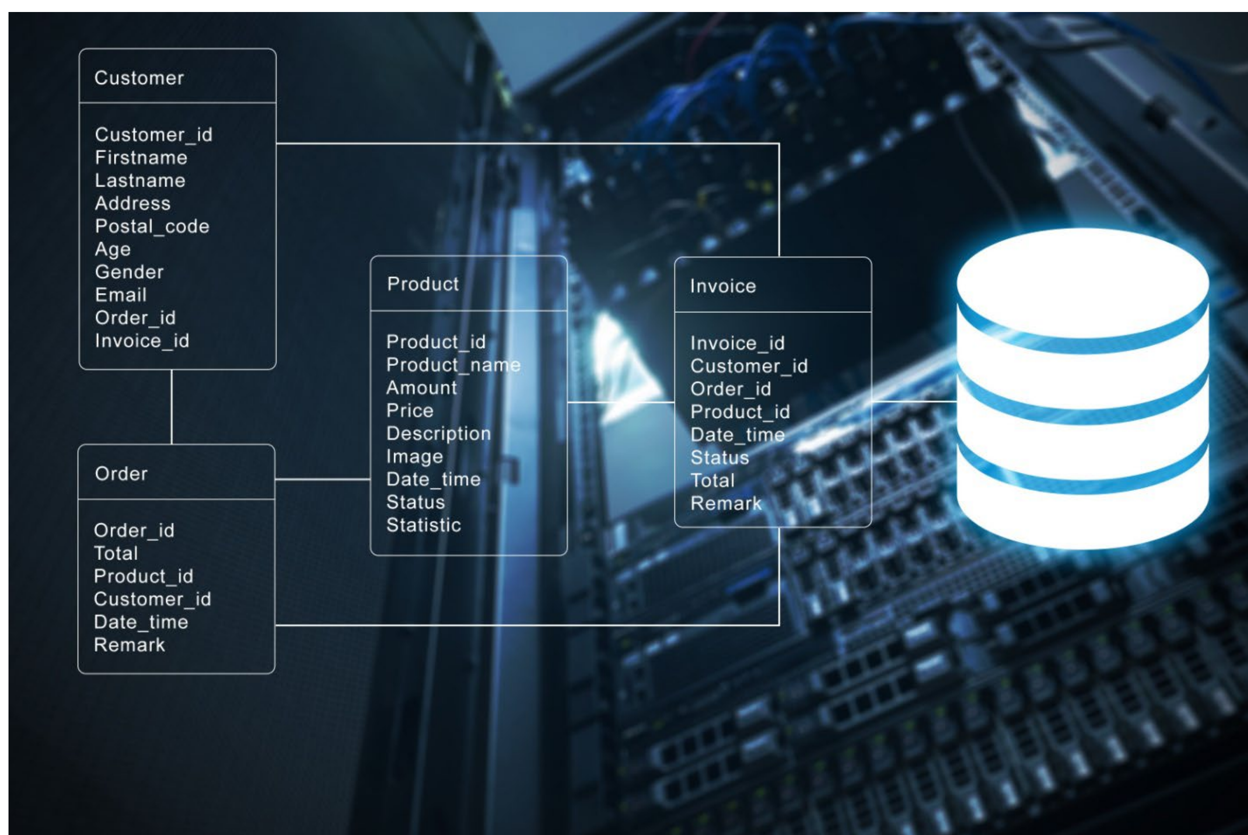
Е. В. БУРЦЕВА, И. П. РАК, А. В. ПЛАТЕНКИН

БАЗЫ ДАННЫХ



Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2023

Министерство науки и высшего образования Российской Федерации

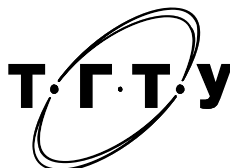
Федеральное государственное бюджетное образовательное учреждение высшего образования «Тамбовский государственный технический университет»

Е. В. БУРЦЕВА, И. П. РАК, А. В. ПЛАТЕНКИН

БАЗЫ ДАННЫХ

Утверждено Ученым советом университета
в качестве учебного пособия для студентов 2 – 4 курсов,
обучающихся по направлению подготовки
09.03.03 «Прикладная информатика»,
очной и заочной форм обучения

Учебное электронное издание



Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2023

УДК 004(075.8)
ББК з973.23я73
Б91

Рецензенты:

Кандидат технических наук, доцент, ведущий инженер-программист
Тамбовского участка ООО «Газпром межрегионгаз Тамбов»
А. Ю. Сенкевич

Доктор технических наук, профессор, проректор по научной работе
ФГБОУ ВО «ТГТУ»
Д. Ю. Муромцев

Бурцева, Е. В.

Б91 Базы данных [Электронный ресурс] : учебное пособие / Е. В. Бурцева, И. П. Рак, А. В. Платенкин. – Тамбов : Издательский центр ФГБОУ ВО «ТГТУ», 2023. – 1 электрон. опт. диск (CD-ROM). – Системные требования : ПК не ниже класса PentiumII ; CD-ROM-дисковод ; 1,2 Mb ; RAM ; Windows 95/98/XP ; мышь. – Загл. с экрана.

ISBN 978-5-8265-2650-7

Рассмотрены типы СУБД PostgreSQL и его версии, вопросы установки, подключения, использования инструментов, расположения файлов сервера базы данных.

Предназначено для студентов 2 – 4 курсов, обучающихся по направлению подготовки 09.03.03 «Прикладная информатика», очной и заочной форм обучения.

УДК 004(075.8)
ББК з973.23я73

Все права на размножение и распространение в любой форме остаются за разработчиком.
Нелегальное копирование и использование данного продукта запрещено.

ISBN 978-5-8265-2650-7

© Федеральное государственное бюджетное образовательное учреждение высшего образования «Тамбовский государственный технический университет» (ФГБОУ ВО «ТГТУ»), 2023

СПИСОК СОКРАЩЕНИЙ

SQL – Structured Query Language, язык структурированных запросов

TOAST – The Oversized-Attribute Storage Technique, Техника хранения больших атрибутов

JSON – JavaScript Object Notation

XML – eXtensible Markup Language, расширяемый язык разметки

RLS – Row-Level Security, Безопасность на уровне строк

MVCC – multiversion concurrency control, управление параллельным доступом посредством многоверсионности

IANA – Управление по присвоению номеров в Интернете

ВВЕДЕНИЕ

PostgreSQL – это мощная система управления базами данных с открытым исходным кодом, пользующаяся популярностью благодаря высокой производительности и стабильности. Благодаря множеству новых функций в своем арсенале PostgreSQL позволяет масштабировать инфраструктуру организаций. С помощью данного пособия вы получите пошаговый, основанный на примерах подход к эффективному администрированию PostgreSQL.

Эта книга поможет вам начать работу со всеми новейшими функциями PostgreSQL, а также поможет вам изучить экосистему базы данных.

Из пособия вы узнаете, как справляться с различными проблемами, с которыми может столкнуться администратор базы данных, такими как создание таблиц, управление представлениями, повышение производительности и защита вашей базы данных. Будут рассмотрены такие темы, как подключение базы данных, регулярное обслуживание базы данных. Затрагиваются такие важные области, как использование сгенерированных столбцов, сжатие TOAST, конфигурация PostgreSQL и многое другое.

1. ПЕРВЫЕ ШАГИ

PostgreSQL – это многофункциональная безопасная система управления базами данных общего назначения. Это сложное программное обеспечение, но каждое путешествие начинается с первого шага.

Мы начнем с вашего первого подключения. Быстро перейдем к включению удаленных пользователей, а оттуда перейдем к получению доступа с помощью инструментов администрирования с графическим интерфейсом.

Мы также представим интерактивный терминал psql, который используется для изучения примеров из книги.

В качестве дополнительной помощи мы включили несколько полезных рецептов, которые могут вам понадобиться для справки. В этой главе мы рассмотрим следующие рецепты:

1.1. ЗНАКОМСТВО С POSTGRESQL14

PostgreSQL – это продвинутая система управления базами данных, доступная на широком спектре платформ. Одним из самых очевидных преимуществ PostgreSQL является то, что она имеет открытый исходный код, а это означает, что у вас есть разрешающая лицензия на установку, использование и распространение PostgreSQL без уплаты каких-либо сборов или роялти. Кроме того, PostgreSQL известна как база данных, которая работает в течение длительного времени и в большинстве случаев практически не требует обслуживания. В целом, PostgreSQL обеспечивает очень низкую совокупную стоимость владения.

PostgreSQL также известен своим огромным набором расширенных функций, разработанных в течение более чем 30 лет непрерывного развития и совершенствования. Первоначально разработанный исследовательской группой баз данных Калифорнийского университета в Беркли, PostgreSQL теперь разрабатывается и поддерживается огромной армией разработчиков и участников.

PostgreSQL имеет следующие основные особенности:

– соответствие стандартам SQL, до SQL: 2016;

– клиент-серверная архитектура;

– высокий уровень параллелизма, в котором чтение и запись не блокируют друг друга;

– легко настраивается и расширяется для любых типов приложений;

– обладает превосходной масштабируемостью и производительностью, а также широкими возможностями настройки;

– предлагает поддержку любых типов моделей данных, таких как реляционные, постреляционные (массивы и вложенные отношения через типы записей), документные (JSON и XML) и ключ/значение.

Что отличает PostgreSQL?

Проект PostgreSQL преследует следующие цели:

– надежное, высококачественное программное обеспечение с поддерживаемым, хорошо комментируемым кодом;

– администрирование, не требующее особого обслуживания, как для индивидуального, так и для корпоративного использования;

– соответствие стандартам SQL, взаимодействие и совместимость;

– производительность, безопасность, и высокая доступность.

Что удивляет многих людей, так это то, что набор функций PostgreSQL больше похож на Oracle или SQLServer, чем на MySQL. Единственная связь между MySQL и PostgreSQL заключается в том, что эти два проекта имеют открытый исходный код; кроме того, особенности и философия почти полностью различны.

PostgreSQL – это система управления базами данных общего назначения. Вы определяете базу данных, которой хотите управлять с ее помощью. PostgreSQL предлагает множество способов работы. Вы можете либо использовать нормализованную модель базы данных, дополненную такими функциями, как массивы и подтипы записей, либо использовать полностью динамическую схему с помощью JSONB и модуля реализующих тип данных hstore. PostgreSQL также позволяет создавать собственные серверные функции на любом из дюжины различных языков.

PostgreSQL обладает высокой расширяемостью, поэтому вы можете добавлять свои собственные типы данных, операторы, типы индексов и функциональные языки. Вы даже можете переопределять различные части системы, используя подключаемые модули для изменения выполнения команд или добавляя новый оптимизатор запросов.

Все эти функции предлагают разработчикам программного обеспечения огромное количество вариантов реализации. Существует множество способов избежать проблем при создании приложений и их поддержке в течение длительного периода времени. К сожалению, в этой книге просто нет места для всех интересных функций для разработчиков; эта книга посвящена администрированию, обслуживанию и резервному копированию.

Кто использует PostgreSQL?

Известные пользователи включают Yandex, MailGroup, Apple, BASF, Skype, McAfee. Начиная с 2010 года количество загрузок PostgreSQL превысило 1 000 000 в год.

PostgreSQL произносится как post-grez-ql. Postgres произносится как post-grez.

Надежность.

PostgreSQL – это надежное высококачественное программное обеспечение, поддерживаемое тестированием как функций, так и параллелизма. По умолчанию база данных предоставляет надежные гарантии записи на диск. Существуют варианты обмена надежности на производительность, хотя они не включены по умолчанию.

Все действия с базой данных выполняются в рамках транзакций, защищенных журналом транзакций, который будет выполнять автоматическое восстановление после сбоя в случае сбоя программного обеспечения.

Базы данных могут быть дополнительно созданы с контрольными суммами блоков данных, чтобы помочь диагностировать аппаратные сбои. Существует множество механизмов резервного копирования, включая полное восстановление на момент времени Point-in-TimeRecovery (PITR). Также доступны различные диагностические инструменты.

Репликация базы данных поддерживается по умолчанию. Синхронная репликация может обеспечить более 5 девяток (99,999%) доступности и защиты данных при правильной настройке и управлении или даже выше при соответствующей избыточности.

Безопасность.

Доступ к PostgreSQL контролируется с помощью правил доступа на основе хоста. Аутентификация является гибкой и подключаемой, что позволяет легко интегрировать ее с любой внешней архитектурой безопасности.

Новейший механизм аутентификации Salted Challenge Response (SCRAM) обеспечивает полную 256-битную защиту.

Доступ с полным шифрованием SSL изначально поддерживается как для доступа пользователей, так и для репликации. Полнофункциональная библиотека криптографических функций доступна для пользователей базы данных.

PostgreSQL предоставляет привилегии доступа на основе ролей для доступа к данным по типу команды. PostgreSQL также обеспечивает безопасность на уровне строк (RLS) для обеспечения конфиденциальности, медицинской и военной безопасности.

Функции могут выполняться с разрешения определителя, в то время как представления могут быть определены с помощью барьеров безопасности, чтобы гарантировать принудительное применение безопасности перед другой обработкой.

Простота использования.

Четкая, полная и точная документация существует как результат процесса разработки, в котором требуются изменения документации. С каждым выпуском происходят сотни небольших изменений, которые сглаживают любые шероховатости использования и предоставляются непосредственно знающими пользователями.

PostgreSQL одинаково работает как на малых, так и на больших системах, а также в разных операционных системах.

Клиентский доступ и драйверы существуют для каждого языка и среды, поэтому нет никаких ограничений на выбор типа среды разработки сейчас или в будущем.

Стандарт SQL соблюдается очень тщательно; нет никакого странного поведения, такого как тихое усечение данных.

Текстовые данные поддерживаются как единый тип данных, который позволяет хранить любые данные от 1 байта до 1 гигабайта. Это хранилище оптимизировано несколькими способами, поэтому 1 байт хранится эффективно, а гораздо большие значения автоматически управляются и сжимаются.

PostgreSQL имеет четкую политику минимизации количества параметров конфигурации, и с каждым выпуском разрабатываются новые способы автоматической настройки параметров.

Расширяемость.

PostgreSQL спроектирован так, чтобы быть расширяемым. Расширения базы данных могут быть легко загружены с помощью CREATE EXTENSION, который автоматизирует проверку версий, зависимостей и других аспектов конфигурации.

PostgreSQL поддерживает определяемые пользователем типы данных, операторы, индексы, функции и языки.

Для PostgreSQL доступно множество расширений, в том числе расширение PostGIS, которое обеспечивает функции Географической Информационной Системы (ГИС).

Производительность и параллелизм.

PostgreSQL может достигать значительно более 1 000 000 операций чтения в секунду.

4-процессорный сервер, производительность которого составляет более 30 000 транзакций записи в секунду при полной надежности, в зависимости от вашего оборудования. С передовым оборудованием возможны еще более высокие уровни производительности.

В PostgreSQL есть усовершенствованный оптимизатор, учитывающий различные типы соединений и использующий статистику пользовательских данных для выбора.

PostgreSQL предоставляет MVCC, который позволяет программам чтения и записи не блокировать друг друга.

В совокупности характеристики производительности PostgreSQL допускают смешанную рабочую нагрузку транзакционных систем и сложных поисковых и аналитических задач. Это важно, потому что это означает, что нам не всегда нужно выгружать наши данные из производственных систем и повторно загружать их в аналитические хранилища данных только для выполнения нескольких специальный запросов. Возможности PostgreSQL делают его предпочтительной базой данных для новых систем, а также правильным долгосрочным выбором почти во всех случаях.

Масштабируемость.

PostgreSQL масштабируется на одном узле до четырех процессорных сокетов. PostgreSQL эффективно запускает до сотен активных сеансов и тысяч подключенных сеансов при использовании пула сеансов. Возможности масштабируемости увеличиваются с каждой новой версией.

PostgreSQL обеспечивает масштабируемость чтения с нескольких узлов с помощью функции горячего ожидания. Масштабируемость записи с несколькими узлами находится в стадии активной разработки.

Модели данных SQL и NoSQL.

PostgreSQL очень точно следует стандарту SQL. Сам SQL не требует использования какого-либо определенного типа модели, поэтому PostgreSQL можно легко использовать для многих типов моделей одновременно в одной и той же базе данных Поскольку PostgreSQL действует как реляционная база данных, мы можем использовать любой уровень денормализации, от полной третьей нормальной формы (3НФ) к более нормализованным моделям звездно-образной схемы. PostgreSQL расширяет реляционную модель, предоставляя массивы, типы строк и типы диапазонов.

База данных, ориентированная на документы, также возможна с использованием типов данных PostgreSQLtext, XML и binaryJSON (JSONB), поддерживаемых индексами, оптимизированными для документов, и возможностями полнотекстового поиска.

Хранилища ключей/значений поддерживаются с помощью расширения hstore.

1.2. СКАЧИВАНИЕ И УСТАНОВКА POSTGRESQL

PostgreSQL – это программное обеспечение со 100% открытым исходным кодом, которое можно свободно использовать, изменять или распространять любым способом по вашему выбору. Berkeley Software Distribution (BSD) – лицензия, хотя и отличается настолько, что теперь известна как Лицензия PostgreSQL (TPL). Вы можете увидеть лицензию здесь:

https://opensource.org/licenses/PostgreSQL.

PostgreSQL уже используется многими пакетами приложений, поэтому вы можете обнаружить, что они уже установлены на ваших серверах. Многие дистрибутивы Linux включают PostgreSQL как часть базовой установки или включают его с установочным диском.

Одна вещь, о которой следует помнить, это то, что включенная версия PostgreSQL может быть не последней версией. Обычно это будет последняя основная версия, которая была доступна на момент публикации версии операционной системы. Обычно нет веских причин придерживаться этого уровня – он не предполагает повышенной стабильности – и более поздние рабочие версии также хорошо поддерживаются различными дистрибутивами Linux, как и более ранние версии.

Если у вас еще нет копии или последней версии, вы можете загрузить исходный код или бинарные пакеты для различных операционных систем

http://www.postgresql.org/download/.

Детали установки значительно различаются от платформы к платформе, и здесь нет никаких особых приемов или рецептов. Просто следуйте инструкциям по установке, и вперед!

Помните, что PostgreSQL – это больше, чем просто основное программное обеспечение. Существует огромное количество веб-сайтов, предлагающих надстройки, расширения и инструменты для PostgreSQL. Вы также найдете множество блоггеров, предлагающих полезные приемы и открытия, которые помогут вам в вашей работе.

Помимо этого, ряд профессиональных компаний может предложить вам помощь, когда вам нужно редактировать.

1.3. ПОДКЛЮЧЕНИЕ К СЕРВЕРУ POSTGRESQL

Подключение к базе данных – это первый опыт работы с PostgreSQL для большинства людей, поэтому опишем это подробно. Давайте сделаем это сейчас и исправим все проблемы, которые возникнут в процессе работы. Помните, что подключение должно быть безопасным, так что нам могут понадобиться некоторые препятствия, чтобы убедиться, что данные, к которым мы хотим получить доступ, безопасны.

Прежде чем мы сможем выполнять команды над базой данных, нам нужно подключиться к серверу базы данных через сеанс пользователя.

Сеансы рассчитаны на продолжительное время, поэтому вы подключаетесь один раз, выполняете множество запросов и, в конце концов, отключаетесь. Во время соединения могут быть небольшие задержки. Это может стать заметным, если вы неоднократно подключаетесь и отключаетесь, поэтому вы можете использовать пулы соединений. Пулы соединений позволяют быстро обслуживать предварительно подключенные сеансы, когда вы хотите восстановить соединение.

Во-первых, кэшируйте свою базу данных. Если вы не знаете, где она находится, у вас, вероятно, возникнут трудности с доступом к ней. Может быть несколько баз данных, и вам нужно знать, к какой из них можно получить доступ, а также иметь полномочия для подключения к ней.

Для подключения к PostgreSQL необходимо указать следующие параметры:

– узел или адрес хоста;
– порт;
– имя базы данных;
– пользователь;
– пароль (или другие средства аутентификации, если таковые имеются).

Для подключения на хосте должен быть запущен сервер PostgreSQL, прослушивающий порт с номером порта. На этом сервере также должна существо-

вать база данных с именем dbname и пользователь с именем user. Хост должен явно разрешить подключения от вашего клиента, и вы также должны пройти аутентификацию, используя метод, указанный сервером, например – указание пароля не сработает, если сервер запросил другую форму аутентификации

Почти все интерфейсы PostgreSQL используют библиотеку libpq. Когда используешь libpq, обработка большинства параметров соединения идентична.

Если вы не укажете указанные параметры, PostgreSQL ищет значения, заданные через переменные среды, в следующих областях:

– PGHOST или PGHOSTADDR;

– PGPORT (установите это значение равным 5432, если оно еще не установлено);

– PGDATABASE;

– PGUSER;

– PGPASSWORD (это определенно не рекомендуется);

Если вы каким-либо образом укажете первые четыре параметра, но не пароль, PostgreSQL выполнит поиск файла пароля.

Некоторые интерфейсы PostgreSQL напрямую используют протокол клиент-сервер, поэтому способы обработки значений по умолчанию могут отличаться.

Каналы деталей подключения также указываются с использованием Универсального идентификатора ресурса (URI) в формате, как показано ниже:

psqlpostgresql://myuser: mypasswd@myhost: 5432/mydb.

Данная форма указывает на то, что мы подключим psql клиентское приложение к серверу PostgreSQL с myhost именем хоста, через порт 5432, cmydb именем базы данных, myuser именем пользователя, mypasswd паролем. Если не указывать пароль URI вам будет предложено ввести пароль.

PostgreSQL – это клиент-серверная база данных. Система, в которой он работает, известна как host. Мы можем получить доступ к серверу PostgreSQL удаленно, через сеть. Однако мы должны указать host, с помощью имени хоста, или адрес хоста, который является IP-адресом. Мы можем указать host как локальный хост если мы хотим установить соединение TCP/IP с той же системой. Часто лучше использовать соединение через Unix сокет, которое будет

осуществлено, если хост начинается с косой черты (/) и предполагается, что это имя является именем каталога (по умолчанию/tmp).

В любой системе может быть более одного сервера базы данных. Каждый сервер базы данных прослушивает ровно один общеизвестный сетевой порт, который не может использоваться совместно серверами в одной системе. Номер порта по умолчанию для PostgreSQL 5432, который был зарегистрирован в Управление по присвоению номеров в Интернете (IANA) и однозначно назначается PostgreSQL.

Номер порта можно использовать для уникальной идентификации определенного сервера базы данных, если таковой существует.

IANA (http://www.iana.org) – это организация, координирующая распределение доступных номеров для различных интернет-протоколов.

Сервер базы данных также иногда называют кластером базы данных, поскольку сервер PostgreSQL позволяет определить одну или несколько баз данных на каждом сервере. Каждый запрос на подключение должен идентифицировать только одну базу данных, определяемую через имя базы данных. При подключении вы сможете видеть только объекты базы данных, созданные в этой базе данных.

Пользователь базы данных используется для идентификации соединения. По умолчанию нет ограничений на количество подключений для конкретного пользователя. В более поздних версиях PostgreSQL пользователей называют ролями входа в систему, хотя многие подсказки напоминают нам о более ранней номенклатуре, и это все еще имеет смысл во многих отношениях. Роль входа – это роль, которой была присвоена привилегия подключения.

Каждое подключение, как правило, каким-либо образом проходит аутентификацию. Это определяется на уровне сервера: аутентификация клиента будет необязательной во время подключения, если администратор настроил сервер так, чтобы это не требовалось.

После того, как соединение установлено, оно может иметь одну активную транзакцию одномоментно и одно полное действие за любое время.

Сервер будет иметь определенное ограничение на количество подключений, которые он может обслуживать, поэтому запрос на подключение может быть отклонен, если сервер перегружен.

Если вы уже подключены к серверу базы данных с помощью psql и хотите подтвердить, что вы подключились к нужному месту и правильным способом, вы можете выполнить некоторые или все из следующих команд. Вот команда, которая показывает текущую базу данных (current_database):

SELECT current_database().

Следующая команда показывает текущий пользовательский идентификатор:

SELECT current_user.

Следующая команда показывает IP-адрес и порт текущего соединения, если только вы не используете Unix сокеты, в этом случае оба значения NULL:

SELECT inet_server_addr(), inet_server_port().

Пароль пользователя недоступен с помощью основных команд SQL по очевидным причинам.

Вы также можете использовать следующий запрос:

SELECT version().

Начиная с PostgreSQL версии 9.1 и далее, вы также можете использовать новую мета-команду psql \conninfo. При этом большая часть предыдущей информации отображается в одной строке:

```
postgres=# \conninfo
You are connected to database postgres, as user postgres, via
socket in /var/run/postgresql, at port 5432.
```

1.4. ВКЛЮЧЕНИЕ ДОСТУПА ДЛЯ СЕТИ/ УДАЛЕННЫХ ПОЛЬЗОВАТЕЛЕЙ

PostgreSQL поставляется в различных дистрибутивах. Во многих из них удаленный доступ изначально отключен из соображений безопасности. Вы можете сделать это быстро, как описано здесь, но на самом деле следует изучить главу о безопасности.

По умолчанию PostgreSQL предоставляет доступ к клиентам, которые подключаются с помощью Unix сокетов, при условии, что пользователь базы данных совпадает с именем пользователя системы.

Шаги следующие:

1. Добавьте или отредактируйте эту строку в вашем файле

postgresql.conf:
listen_addresses='*'

2. Добавьте следующую строку в качестве первой строки pg_hba.conf разрешить доступ ко всем базам данных для всех пользователей с зашифрованным паролем:

TYPE DATABASE USER CIDR-ADDRESS METHOD
host all all 0.0.0.0/0 scram-sha-256

После изменения listen_addresses, перезапустить сервер PostgreSQL.

Параметры listen_addresses указывают, какой IP-адрес следует прослушивать. Это позволяет вам гибко включать и отключать прослушивание интерфейсов нескольких Карт сетевого интерфейса (сетевые карты) или виртуальные сети в той же системе. В большинстве случаев мы хотим принимать соединения со всеми сетевыми картами, поэтому мы используем *, что означает все IP-адреса.

Файл pg_hba.conf содержит набор правил аутентификации на основе хоста. Каждое правило рассматривается последовательно до тех пор, пока не сработает одно правило или пока попытка не будет специально отклонена с помощью правил отклонения.

Предыдущее правило означает, что удаленное соединение, которое указывает любого пользователя или базу данных на любом IP-адресе, будет запрошено для аутентификации с использованием пароля, зашифрованного SCRAM-SHA-256. Ниже приведены параметры, необходимые для паролей, зашифрованных SCRAM-SHA-256:

Тип: имя host с разрешением – удаленное соединение.

База данных: означает для всех баз данных. Разрешения даются для имен с точным совпадением, за исключением если перед именем ставится плюс (+), в этом случае имеют в виду групповую роль, а не роль одного пользователя. Вы также можете указать список пользователей, разделенных запятыми, или использовать символ @ чтобы включить файл со списком пользователей. Вы даже можете указать того же пользователя, чтобы правило совпадало, когда вы указываете одно и то же имя для пользователя и базы данных.

Пользователь: означает для всех пользователей. Разрешения даются для имен с точным совпадением, за исключением случаев, когда перед именем ставится символ (+), в этом случае имеются в виду групповую роль, а не один пользователь. Можно указать список пользователей, разделенных запятыми, или использовать символ @ для включения файла со списком пользователей.

CIDR-ADDRESS: состоит из двух частей: IP-адреса и маски подсети. Маска подсети определяется как количество старших битов IP-адреса, составляющих маску. Таким образом, /0означает 0 бит IP-адреса, так что все IP-адреса будут совпадать. Например, 192.168.0.0/24 означало бы соответствие первым 24 битам, поэтому любая форма IP-адреса 192.168.0.x будет им соответствовать. Вы также можете использовать samenet или samehost.

Метод: SCRAM-SHA-256 означает, что PostgreSQL попросит клиента предоставить пароль, зашифрованный с помощью SCRAM-SHA-256. Другим распространенным параметром является доверие, что фактически означает отсутствие аутентификации. Другие методы аутентификации включают GSSAPI, SSPI, LDAP, RADIUS и PAM. Соединения PostgreSQL также могут быть выполнены с использованием SSL, и в этом случае клиентские SSL-сертификаты обеспечивают аутентификацию.

Не используйте метод аутентификации через пароль в pg_gba.conf, так как пароль отправляете открытым текстом. Это не является серьезной проблемой безопасности, если ваше соединение зашифровано с помощью SSL, в любом случае обычно нет недостатков с SCRAM-SHA-256, и у вас есть дополнительная безопасность для соединений без SSL.

1.5. ИСПОЛЬЗОВАНИЕ ИНСТРУМЕНТА PGADMIN4GUI

Системные администраторы часто запрашивают графические инструменты администрирования. PostgreSQL имеет ряд опций инструментов. В этой книге мы рассмотрим pgAdmin4.

pgAdmin4 – это клиентское приложение, которое отправляет и получает SQL в и из PostgreSQL, отображая результаты для вас. Клиент администратора может получить доступ ко многим серверам баз данных, что позволяет вам управлять несколькими серверами. Инструмент работает как в автономном режиме приложения, так и в веб-браузерах.

pgAdmin4 обычно называется просто pgAdmin. У 4 в конце длинная история, но это не так важно. Это не уровень выпуска; pgAdmin4 заменяет более раннюю rpgAdmin3. Когда вы запустите pgAdmin, вам будет предложено зарегистрировать новый сервер.

Дайте вашему серверу имя на Общий вкладке, а затем щелкните Связь и заполните пять основных параметров подключения, а также другую информацию. Сохраните пароль (рис. 1.1).

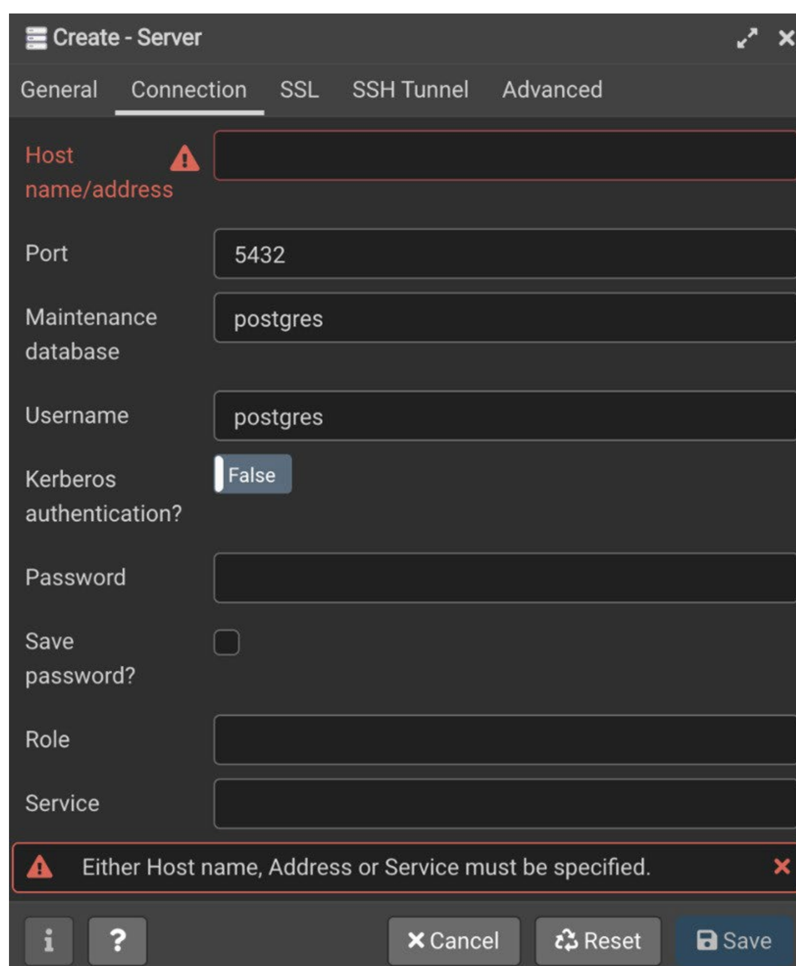


Рис. 1.1. Свойства подключения к серверу

Если у вас много серверов базы данных, вы можете сгруппировать их вместе. Предлагаем держать все реплицированные серверы вместе в одной группе серверов. Дайте каждому серверу разумное имя.

После добавления сервера вы можете подключиться к нему и отобразить информацию о нем.

Экран по умолчанию – панель мониторинга, на которой представлено несколько интересных графиков, основанных на данных, которые она извлекает с сервера. Это очень полезно, поэтому перейдите на вкладку Статистика.

После этого вы получите доступ к главному экрану браузера с деревом объектов слева и статистикой справа, как показано на следующем снимке экрана (рис. 1.2).

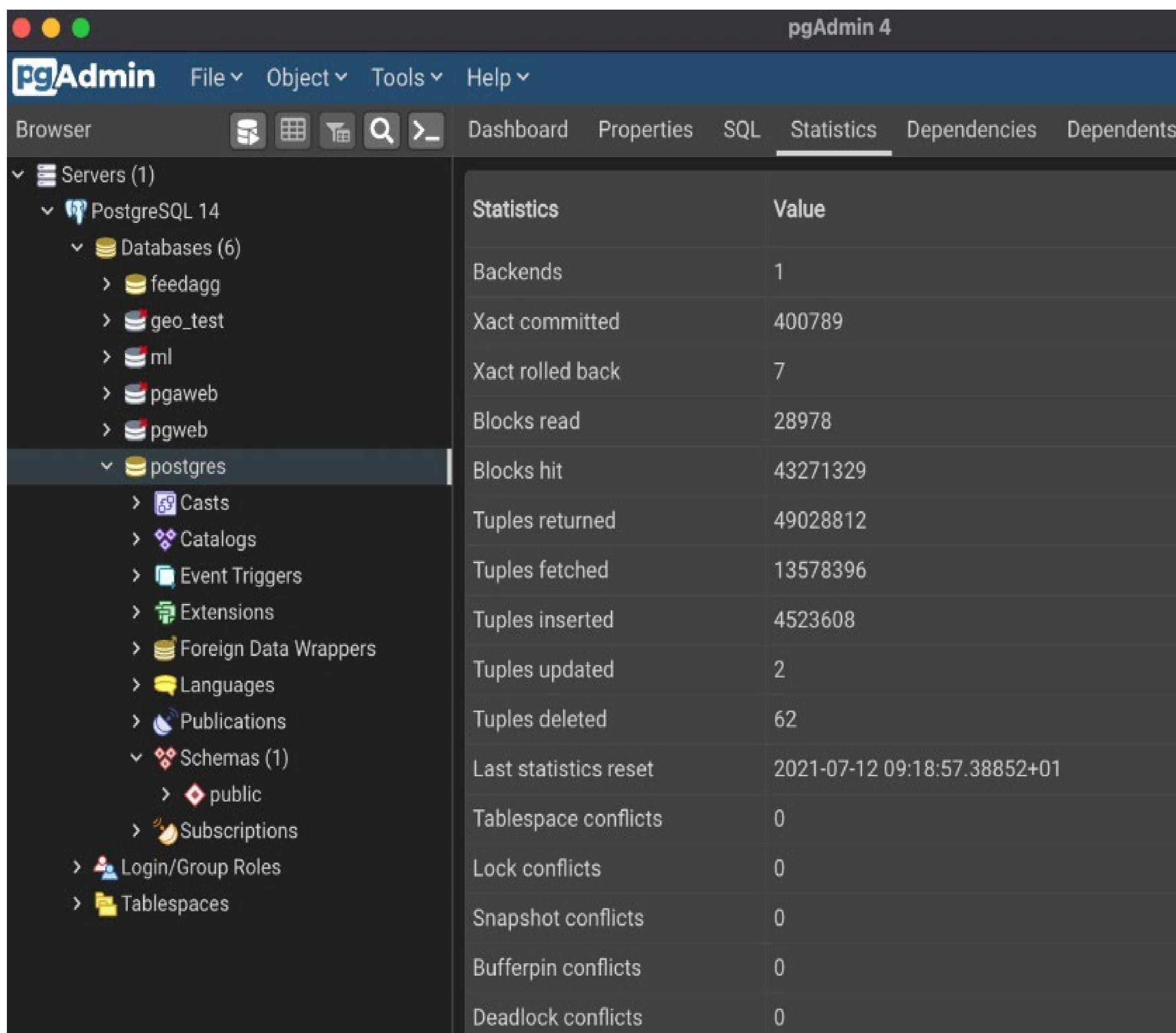


Рис. 1.2. Представление pgAdminTreeView с вкладкой Статистика

pgAdmin легко отображает большую часть данных, доступных в PostgreSQL. Информация является контекстнозависимой, что позволяет вам перемещаться и видеть все быстро и легко. Информация не обновляется динамически, это происходит только тогда, когда вы нажимаете кнопку для обновления, поэтому имейте это в виду при использовании приложения.

pgAdmin также предоставляет Grant Wizard. Это полезно для администраторов баз данных для обзора и немедленного обслуживания (рис. 1.3).

Инструмент запроса pgAdmin позволяет вам контролировать несколько активных сеансов. Инструмент запроса имеет привлекательную функцию визуального объяснения, которая отображает план объяснения для вашего запроса (рис. 1.4).

pgAdmin предоставляет широкий спектр функций, многие из которых предоставляются различными инструментами. Это дает нам возможность выбирать, какой из этих инструментов нам нужен. По многим причинам лучше всего использовать правильный инструмент для правильной работы, и это всегда зависит от знаний, опыта и личного вкуса.

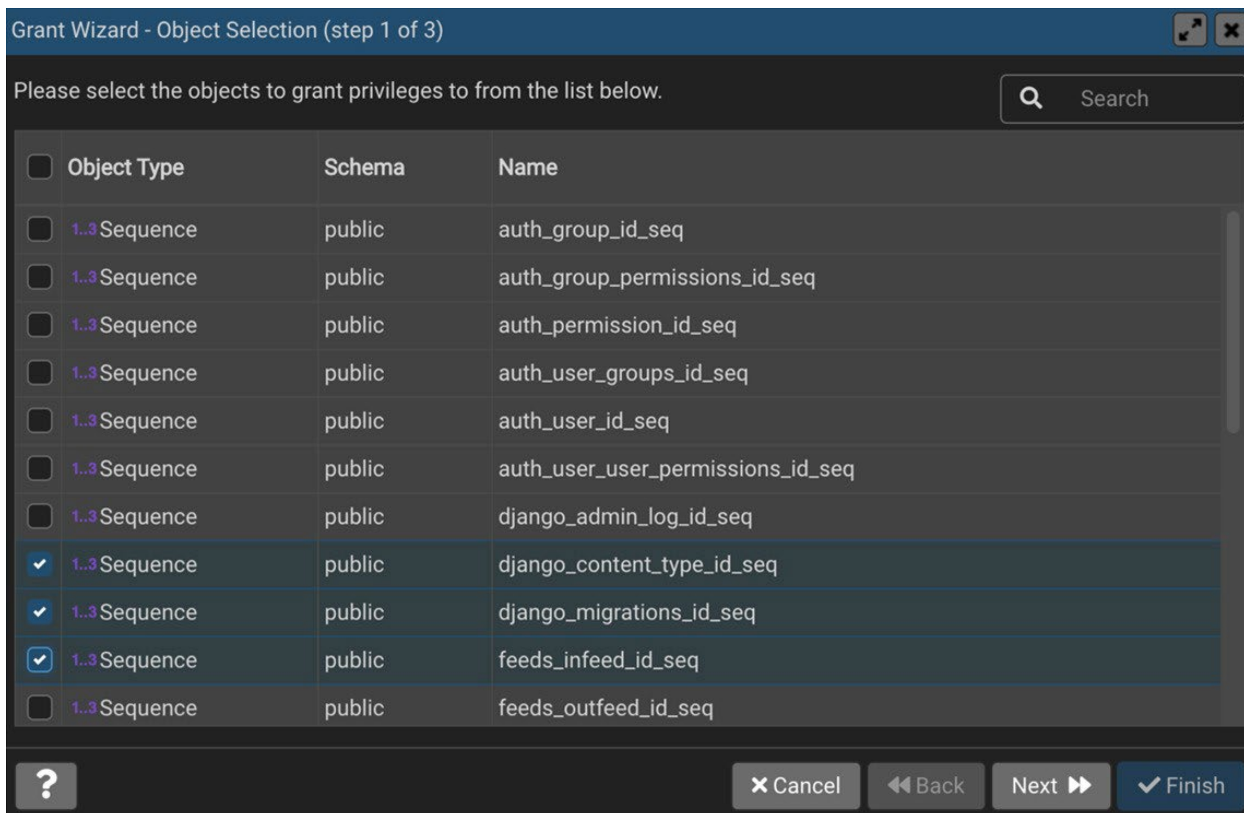


Рис. 1.3 Grant Wizard

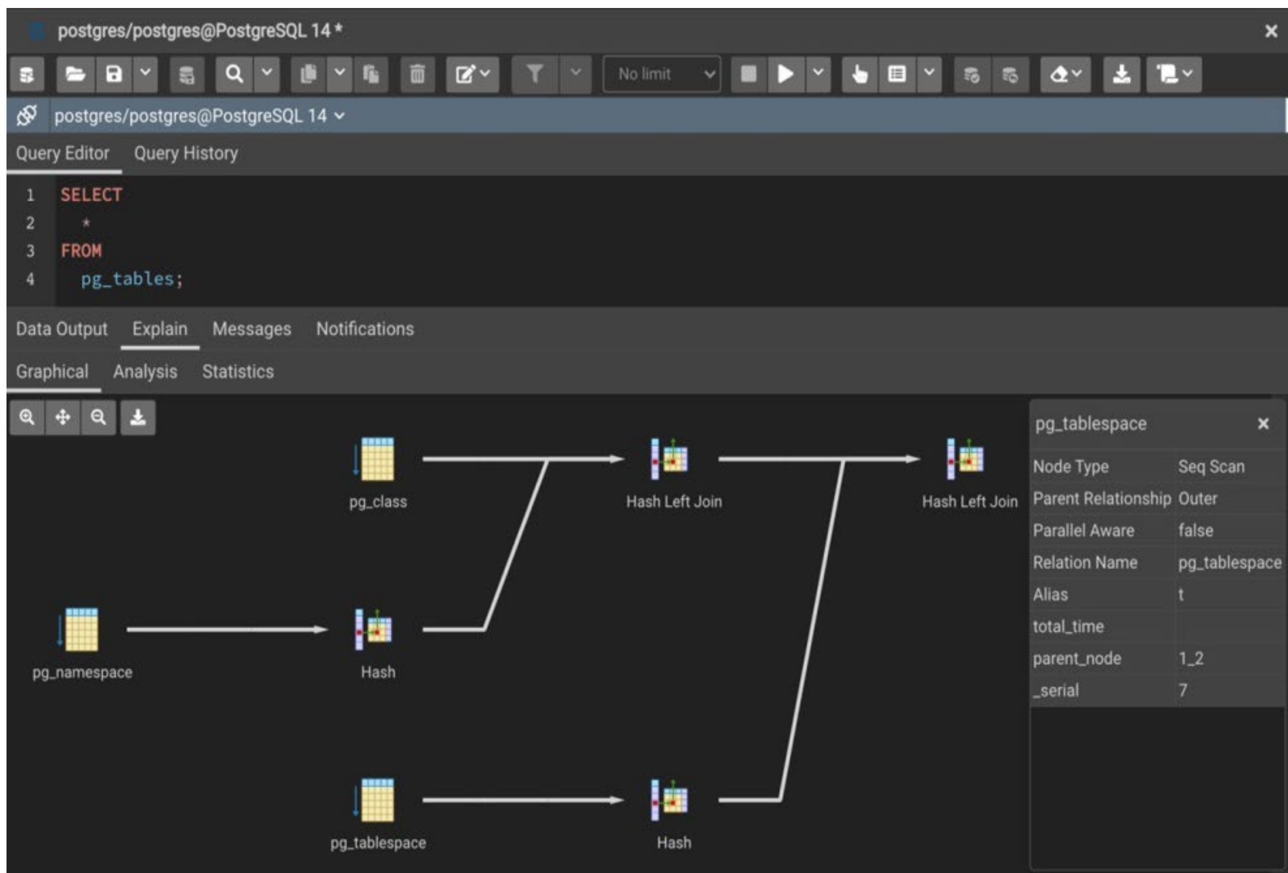


Рис. 1.4. Функция визуального объяснения

pgAdmin отправляет SQL на сервер PostgreSQL и отображает результаты быстро и легко. Для выполнения небольших задач DBA это идеально. Как вы могли догадаться из этих комментариев, мы не рекомендуем инструменты с графическим интерфейсом для каждой задачи.

Написание сценариев – важная техника для администраторов баз данных. Вы сохраняете копию выполненной задачи и можете отредактировать ее и повторно отправить в случае возникновения проблем. Также легко поместить все задачи сценария в одну транзакцию, что невозможно при использовании текущих инструментов с графическим интерфейсом. Для написания сценариев рекомендуется использовать утилиту psql, которая имеет множество дополнительных функций.

Хотя рекомендуется использовать psql в качестве скриптового инструмента, многие люди находят его удобным в качестве инструмента запросов. Некоторым людям это может показаться странным, и они предполагают, что это выбор только для экспертов. Две замечательные функции psql – это онлайн-

справка по SQL и функция завершения работы с вкладками, которая позволяет быстро создавать SQL без необходимости запоминать синтаксис.

pgAdmin предоставляет инструмент PSQL, который позволяет запускать psql вместе с pgAdmin.

Это отличное нововведение, которое позволяет вам использовать возможности графического интерфейса наряду с возможностями psql.

2. ИЗУЧЕНИЕ БАЗЫ ДАННЫХ

Чтобы понять PostgreSQL, нужно увидеть, как он используется. Пустая база данных похожа на город-призрак без домов.

Пока будем считать, что у вас уже есть база данных. Существует более тысячи книг о том, как создать собственную базу данных с нуля. Итак, здесь мы стремимся помочь людям, которые все еще учатся использовать систему управления базой данных PostgreSQL с помощью простых процедур для изучения базы данных.

Лучший способ начать – это задать несколько простых вопросов, чтобы сориентироваться и начать процесс понимания.

В этой главе будут рассмотрены следующие вопросы:

Что это за сервер?

Какая версия сервера?

Каково время работы сервера?

Расположение файлов сервера базы данных.

Местонахождение журнала сообщений сервера базы данных.

Поиск системного идентификатора базы данных.

Список баз данных на сервере базы данных.

Сколько таблиц находится в базе данных?

2.1. ТИП СУБД POSTGRESQL И ЕЕ ВЕРСИИ

PostgreSQL – система управления объектно-реляционной базой данных (СУБД) с открытым исходным кодом распространяется по разрешающей лицензии и разрабатывается активным сообществом.

Существует ряд сервисов и программного обеспечения, связанных с PostgreSQL (https://wiki.postgresql.org/wiki/PostgreSQL_derived_databases), как с открытым исходным кодом, так и без него, которые предоставляются другими компаниями-разработчиками программного обеспечения. Здесь мы обсудим, как распознать, какой из них вы используете.

Не всегда явно можно определить вариант PostgreSQL по названию; многие продукты и услуги, использующие PostgreSQL, содержат слово PostgreSQL.

Однако, если вам необходимо проверить документацию или приобрести такие услуги, как поддержка или консультации, вам необходимо точно определить тип вашего сервера, поскольку доступные варианты будут различаться.

К сожалению, нет ни одной функции или параметра, которые работали бы на каждом варианте PostgreSQL и в то же время могли бы ответить на этот вопрос. Самое близкое, что вы можете использовать – это функция version(), которая возвращает текстовое описание версии, которую вы используете, включая (но не ограничиваясь этим) номер версии.

В некоторых случаях этого достаточно, но в других случаях вам придется определить конкретную версию по другим подсказкам, таким как:

– номер версии для стабильных выпусков из сообщества PostgreSQL двух видов: либо X.Y (с X = 10 или выше), либо X.YZ (до X = 9). Дополнительный номер обычно указывает на то, что вы используете какой-то вариант PostgreSQL.

– наличие определенных объектов, которые доступны только в определенной версии, например, по наличию расширений.

PostgreSQL имеет внутренние номера версий для формата файла данных, макета каталога базы данных и формата восстановления после сбоя. Каждый из них проверяется во время работы сервера, чтобы гарантировать, что данные не будут повреждены. PostgreSQL не изменяет эти внутренние форматы для отдельного дополнительного выпуска; они меняются только в основных выпусках.

С точки зрения пользователя каждый выпуск отличается поведением сервера. Если вы хорошо знаете свое приложение, т.е. возможность оценить различия, просто прочитав примечания к выпуску для каждой версии. Во многих случаях повторное тестирование приложения является наиболее безопасным.

Если у вас возникнут какие-либо общие проблемы, связанные с установкой и конфигурированием вашей базы данных, вам необходимо дважды проверить, какая версия сервера у вас установлена. Это поможет вам сообщить об ошибке или обратиться к правильной версии руководства.

Чтобы узнать версию, можно обратиться напрямую к серверу БД:

– подключитесь к базе данных и выполните следующую команду:

```
postgres=# SELECTversion();
```

– вы получите ответ, который выглядит примерно так:

PostgreSQL 14.0 (Debian 14.0-1.pgdg100+1) on x86_64-pclinux-gnu, compiled by gcc (Debian 8.3.0-6) 8.3.0, 64-bit

Другой способ проверки номера версии в вашей программе заключается в следующем:

postgres=#SHOWserver_version;

В этом примере версия будет представлена в текстовом виде, поэтому если вам понадобится числовое значение, которое легче воспринимать и сравнивать, выполняете эту команду:

postgres=#SHOWserver_version_num;

Другая альтернатива – утилиты командной строки, такие как:

bash # psql --version
psql (PostgreSQL) 14.0 (Debian 14.0-1.pgdg100+1

Однако помните, что здесь отображается номер версии программного обеспечения клиента, который может отличаться от номера версии программного обеспечения сервера. Обычно об этом вам сообщают, чтобы вы знали.

Текущий формат версии сервера PostgreSQL состоит из двух чисел; первое число указывает основной выпуск, а второе обозначает последующие выпуски обслуживания для этого основного выпуска. Обычно при обсуждении поддерживаемых функций упоминается только основная версия, поскольку они остаются неизменными в поддерживаемой версии.

Например, 14.0 – это первый выпуск PostgreSQL 14, а последующие отладочные выпуски будут 14.1, 14.2, 14.3 и так далее.

Для каждого основного выпуска существует отдельная версия руководства, поскольку набор функций неодинаков. Если что-то работает не так, как вы думаете, убедитесь, что вы используете правильную версию руководства.

До выпуска 10 в PostgreSQL использовалась нумерация из трех частей, что означало, что набор функций и совместимость относились к первым двум числам, а отладочные выпуски обозначались третьим номером. Например, версия 9.6 содержала больше дополнительных функций и изменений совместимости по сравнению с версией 9.5; версия 9.6.0 была первоначальным выпуском 9.6, а версия 9.6.1 – последним выпуском обслуживания.

Политика поддержки выпуска для PostgreSQL доступна вhttp://www.postgresql.org/support/versioning/. В этой статье объясняется, что каждый выпуск будет поддерживаться в течение 5 лет. Поскольку выпускается одна основная версия в год, это означает 5 основных выпусков.

Поддержка всех выпусков до версии 9.6 включительно закончилась в сентябре 2021 года. Таким образом, к настоящему моменту будет поддерживаться только PostgreSQL 10 и более поздние версии. Более ранние версии по-прежнему надежны, хотя в этих выпусках отсутствуют многие функции производительности и корпоративные функции.

Version	**Release date**	**Last supported date**
PostgreSQL 10	September 2017	November 2022
PostgreSQL 11	October 2018	November 2023
PostgreSQL 12	October 2019	November 2024
PostgreSQL 13	September 2020	November 2025
PostgreSQL 14	September 2021	September 2026

Рис. 2.1. Таблица, показывающая версию PostgreSQL, и даты выпуска

2.2. ВРЕМЯ БЕЗОТКАЗНОЙ РАБОТЫ

Вам может быть интересно, сколько времени прошло с момента запуска сервера?

Например, вы можете захотеть убедиться, что не было сбоя сервера, если ваш сервер не отслеживается, или посмотреть, когда сервер был в последний раз перезапущен, например, для изменения конфигурации.

Выполните следующий SQL из любого интерфейса:

```
postgres=# SELECT date_trunc('second', current_timestamp –
pg_ postmaster_start_time()) as uptime;
```

Вы должны получить вывод следующим образом:

```
uptime
------------------------------------
2 days 02:48:0
```

Postgres хранит время старта сервера, поэтому мы можем узнать его следующим образом:

```
postgres=# SELECT pg_postmaster_start_time();
pg_postmaster_start_time
-----------------------------------------------
2021-10-01 19:37:41.389134+00
```

Затем мы можем написать SQL-запрос для получения времени безотказной работы, например:

```
postgres=# SELECT current_timestamp – pg_postmaster_start_
time();

?column?
-----------------
02:59:18.925917
```

Наконец, мы можем применить некоторое форматирование:

```
postgres=# SELECT date_trunc('second', current_timestamp – pg_
postmaster_start_time()) as uptime;
uptime
----------
03:00:26
```

2.3. РАСПОЛОЖЕНИЕ ФАЙЛОВ СЕРВЕРА БАЗЫ ДАННЫХ

Файлы сервера базы данных изначально хранятся в расположении, называемом каталог данных. Дополнительные файлы данных также могут храниться в табличных пространствах, если они существуют.

В этом рецепте вы узнаете, как найти расположение этих каталогов на заданном сервере базы данных.

Вам потребуется получить доступ из операционной системы к системе базы данных, которую называют платформой, на которой работает база данных.

Если есть возможность подключиться с помощью утилиты psql, можно использовать эту команду:

```
postgres=# SHOW data_directory;
data_directory
----------------------
/var/lib/pgsql/data/
```

Если нет, то в зависимости от операционной системы по умолчанию данные расположены в следующих каталогах:

Системы Debian или Ubuntu:/var/lib/postgresql/MAJOR_RELEASE/main
RedHatRHEL, CentOS и Fedora:/var/lib/pgsql/data/
Windows:C:\ProgramFiles\PostgreSQL\MAJOR_RELEASE\data
MAJOR_RELEASE состоит из одного числа (для выпусков 10 и выше) или двух (для выпусков до 9.6).

В системах Debian или Ubuntu файлы конфигурации находятся в/etc/postgresql/MAJOR_RELEASE/main/, где mainэто просто имя сервера базы данных. Другие имена возможны. Ради простоты рассматривается, что у вас есть только одна установка сервера, хотя смысл включения номера версии и имени сервера базы данных в качестве компонентов пути к каталогу состоит в том, чтобы позволить нескольким серверам баз данных сосуществовать на одном хосте.

Информация для каждого сервера включает следующее:

- номер основного выпуска;
- порт;
- статус (например, онлайн и не работает);
- каталог данных;
- файл логов;

Утилита pg_listclusters является частью пакета postgresql-common Debian/Ubuntu, который предоставляет структуру, в соответствии с которой можно устанавливать несколько версий PostgreSQL и поддерживать несколько кластеров одновременно.

В пакетах, распространяемых с RedHat RHEL, CentOS и Fedora, каталог данных по умолчанию также содержит файлы конфигурации (*.conf) по умолчанию. Однако обратите внимание, что пакеты, распространяемые сообществом PostgreSQL, используют другое расположение по умолчанию: /var/lib/pgsql/MAJOR_RELEASE/data/.

Опять же, это просто расположение по умолчанию. Вы можете создавать дополнительные каталоги данных, используя утилиту initdb.

Утилита initdb заполняет заданный каталог данных исходным содержимым. Каталог будет создан для удобства, если он отсутствует, но в целях безопасности утилита остановится, если каталог данных не пуст. Утилита initdb

считает имя каталога данных из переменной среды PGDATA, если не используется параметр командной строки -d.

Найдя каталог данных, вы можете найти файлы, составляющие сервер базы данных PostgreSQL. Схема выглядит следующим образом (рис. 2.2).

Подкаталог	Назначение
base	Это основное хранилище таблиц. Под этим каталогом каждая база данных имеет свой собственный каталог, в котором расположены файлы для каждой таблицы базы данных или индекса.
global	Таблицы, которые являются общими для всех баз данных, включая список баз данных.
pg_commit_ts	Здесь мы храним данные временной метки фиксации транзакции (начиная с версии 9.5 и далее).
pg_dynshmem	Включает в себя информацию о динамической общей памяти (начиная с версии 9.4 и далее).
pg_logical	Включает в себя данные о состоянии логического декодирования.
pg_multixact	Включает файлы, используемые для общих блокировок на уровне строк.
pg_notify	Включает в себя файлы состояния ПРОСЛУШИВАНИЯ/ УВЕДОМЛЕНИЯ
pg_replslot	Включает информацию о слотах репликации (начиная с 9.4 и далее).
pg_serial	Включает в себя информацию о совершенных сериализуемых транзакциях.
pg_snapshots	Включает в себя информацию о завершенных сериализуемых транзакциях.
pg_stat	Включает в себя постоянные статистические данные.
pg_stat_tmp	Включает в себя данные временной статистики.
pg_subtrans	Включает данные о статусе субтранзакции.
pg_tblspc	Включает символические ссылки на каталоги табличного пространства.
pg_twophase	Включает в себя главные файлы для подготовленных транзакций.
pg_wal	Включает в себя журнал транзакций или журнал предварительной записи (WAL) (ранее pgxlog).
pg_xact	Включает файлы статуса транзакции (ранее pgclog).

Рис. 2.2. Содержимое каталога данных PostgreSQL

Ни один из вышеупомянутых каталогов не содержит файлов, которые могут быть изменены пользователем, и ни один из файлов не должен быть удален вручную для экономии места или по какой-либо другой причине. Не трогайте его, потому что если испортить его или удалить то, возможно, не будет возможности его исправить!

Единственное, к чему вам разрешено прикасаться, – это файлы конфигурации *.conf и файлы журнала сообщений сервера. Файлы журнала сообщений сервера могут находиться в каталоге данных, а могут и не находиться.

Логи сообщения сервера базы данных представляют собой записи сообщений, записанных сервером базы данных. Это первое место, где можно проверить наличие проблем с сервером, и место для регулярной проверки.

Этот журнал будет включать сообщения, которые будут выглядеть примерно следующим образом:

2021-09-01 19:37:41 GMT [2507-1] LOG: database system was shut down at 2021-09-01 19:37:38 GMT

2021-09-01 19:37:41 GMT [2506-1] LOG: database system is ready to accept connections

Журнал сервера может находиться в нескольких разных местах, поэтому давайте сначала перечислим их все, чтобы мы могли найти журнал или решить, где хотите его разместить:

– журнал сервера может находиться в одном из подкаталогов основного каталога данных.

– может находиться в каталоге в другом месте файловой системы.

– может быть перенаправлен в системный журнал.

– журнал сервера может вообще отсутствовать. В этом случае необходимо добавить журнал.

Если процесс не перенаправлен в системный журнал, журнал сервера состоит из одного или нескольких файлов. Вы можете изменить имена этих файлов, поэтому они не всегда могут быть одинаковыми для каждой системы.

Расположение журнала сервера по умолчанию:

Системы Debian или Ubuntu:/var/log/postgresql.

RedHat, RHEL, CentOSиFedora:/var/lib/pgsql/data/pg_log.

Windows: сообщения отправляются в WindowsEventLog.

Текущий файл журнала сервера называется postgresql-MAJOR_ RELEASE-SERVER.log, где SERVER имя сервера (по умолчанию, основного), иMAJOR_RELEASE указывает основной выпуск сервера, например, 9.6 или 11. Приме ppostgresql-14-main.log, в то время как более старые файлы журнала пронумерованы как postgresql-14-main.log.1. Чем больше конечный номер, тем старше файл, поскольку они ротируются утилитой logrotate.

Журнал сервера – это просто файл, в который записываются сообщения с сервера. Каждое сообщение имеет уровень важности, наиболее типичным из которых является LOG, хотя есть и другие, как показано в следующей таблице (рис. 2.3).

Важность для сервера	**Значение**	**Важность по журналу**	**WindowsEventI.og**
DEBUG 1 to DEBUG 5	Включает в себя внутреннюю диагностику	DEBUG	INFORMATION
INTO	Вывод команды для пользователя	INFO	INFORMATION
NOTICE	Полезная информация	NOTICE	INFORMATION
WARNING	Предупреждает о вероятных проблемах	NOTICE	WARNING
ERROR	Текущая команда, которая прервана	WARNING	ERROR
LOG	Полезно для системных администраторов	INFO	INFORMATION
FATAL	Событие, которое отключает только один сеанс, только для сеанса	ERR	ERROR
PANIC	Событие приводит к сбою сервера	CRIT	ERROR

Рис. 2.3. Уровни серьезности сообщений PostgreSQL

Берегись FATAL и PANIC. Этих событий не должно происходить в большинстве случаев при нормальной работе сервера, за исключением некоторых случаев, связанных с репликацией.

Вы можете настроить количество сообщений, отображаемых в журнале, изменив log_min_messages в параметрах сервера. Вы также можете изменить объем информации, отображаемой перед событием, изменив параметр

log_error_verbosity. Если сообщения отправляются в стандартный лог-файл, тогда каждая строка лога будет иметь префикс полезной информации, которой также может управлять системный администратор через параметр с именем log_line_prefix.

Вы также можете изменить что и сколько записывается в журналы, изменив другие параметры, такие как log_statements, log_checkpoints, log_connections/log_disconnections, log_verbosity, и log_lock_waits.

Параметр log_destination определяет, где хранятся сообщения журнала. Допустимыми значениями являются stderr, csvlog, syslog и eventlog (последнее доступно только в Windows).

Сборщик журналов – это фоновый процесс, который записывает в файл журнала все, что сервер PostgreSQL выводит в stderr. Это, вероятно, самый надежный способ протоколирования сообщений в случае возникновения проблем, поскольку он зависит от меньшего количества служб.

Ротацией журнала можно управлять с помощью таких настроек, как log_rotation_age и log_ rotation_size, если вы используете сборщик журналов. В качестве альтернативы, можно настроить утилиту logrotate для выполнения ротации журналов, которая используется по умолчанию в системах Debianи Ubuntu.

2.4. СПИСОК БАЗ ДАННЫХ НА СЕРВЕРЕ БАЗЫ ДАННЫХ

Когда мы подключаемся к PostgreSQL, мы всегда подключаемся только к одной конкретной базе данных на любом сервере базы данных. Если на одном сервере много баз данных, это может привести к путанице, поэтому иногда вы можете просто захотеть выяснить, какие базы данных являются частью сервера базы данных.

Если у вас есть доступ к psql, вы можете ввести следующую команду:

```
bash $ psql -l
ListofdatabasesName | Owner | Encoding | Collate | Ctype |
Accessprivileges
-----------+--------+----------+------------+------------+---
----------------
postgres | sriggs | UTF8 | en_GB.UTF-8 | en_GB.UTF-8 |
template0 | sriggs | UTF8 | en_GB.UTF-8 | en_GB.UTF-8 |
```

```
=c/sriggs +
| | | | |
sriggs=CTc/sriggs
template1 | sriggs | UTF8 | en_GB.UTF-8 | en_GB.UTF-8 |
=c/sriggs +
| | | | |
sriggs=CTc/sriggs
(3 rows)
```

Вы также можете получить ту же информацию во время запуска psql, просто набрав \l.

Информация, которую мы только что просмотрели, хранится в таблице каталога PostgreSQL с именем pg_database. Мы можем выполнить SQL-запрос непосредственно к этой таблице из любого соединения, чтобы получить более простой результат, как показано ниже:

```
postgres=# selectdatnamefrompg_database;
datname
-----------
template1
template0
postgres
(3 rows)
```

PostgreSQL запускается изначально с тремя базами данных: template0, template1, и postgres. Основная база данных пользователей postgres.

Вы также можете создавать собственные базы данных, например:

CREATE DATABASE cookbook;

Вы можете сделать то же самое из командной строки, используя следующее выражение:

```
bash $ createdbcookbook
```

С этого момента мы будем запускать наши примеры в базе данных cookbook.

Когда вы создаете еще одну базу данных, вы фактически берете копию существующей базы данных.

Базы данных template0 и template1 называют шаблонами базы данных.

Базу данных template1 можно изменить, чтобы использовать ее в качестве шаблона для любых новых баз данных, которые вы создаете. База данных template0 работает так, что, когда меняется template1, у вас все еще есть первоначальная копия, на которую можно вернуться; template1 можно удалить и воссоздать из template0.

Вы можете удалить базу данных с именем postgres. Но не надо этого делать, так как в ней хранятся системные параметры и настройки СУБД. Точно также не пытайтесь менять template0, потому что система не разрешит этого сделать, template0 можно использовать только в качестве шаблона. С другой стороны, база данных template1 создана для изменения, так что не стесняйтесь изменять ее.

2.5. ТАБЛИЦЫ В БАЗЕ ДАННЫХ

Количество таблиц в реляционной базе данных является хорошей мерой сложности базы данных, поэтому это простой способ познакомиться с любой базой данных. Чтобы узнать число таблиц в любом интерфейсе введите следующую команду SQL:

```
SELECT count(*) FROM information_schema.tables WHERE table_schema
NOT IN ('information_schema','pg_catalog');
```

Можно самостоятельно просмотреть список таблиц и решить, является ли список маленьким или большим.

Bpsql, вы можете просмотреть свои собственные таблицы с помощью следующей команды:

```
$ psql -c "\d"
Listofrelations
Schema | Name | Type | Owner
--------+----------+-------+----------
public | accounts | table | postgres
public | branches | table | postgres
```

BpgAdmin4, вы можете увидеть таблицы в дереве слева, как показано на следующем снимке экрана (рис. 2.4).

PostgreSQL хранит информацию о базе данных в таблицах каталогов. Они описывают каждый аспект определения базы данных. В схеме хранится основной набор таблиц каталога, называемый pg_catalog. Существует второй набор объектов каталога, называемый информационная схема, который является стандартным для SQL способом доступа к информации в реляционной базе данных.

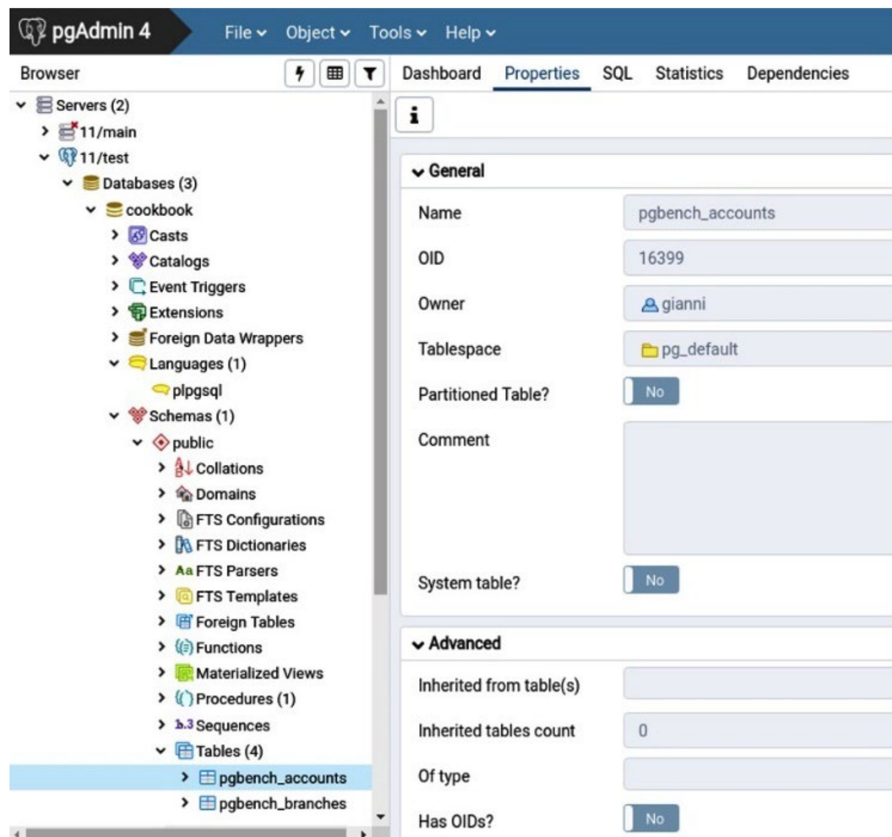


Рис. 2.4. Древовидное представление объектов базы данных в pgAdmin

Чтобы исключить обе эти схемы из нашего запроса и избежать подсчета неиспользуемых объектов необходимо использовать фразы NOT IN и WHERE как в примере выше.

Исключение разделов из подсчета более сложно. Информационная схема показывает разделы как те же самые таблицы, что верно для PostgreSQL, но несколько вводит в заблуждение. Допустим, мы хотим исключить таблицы, которые также являются разделами. Разделы отмечены в pg_catalog.pg_class колонкой типа BOOLEAN. Если использовать pg_class, мы также должны исключить не-таблицы и убедиться, что мы не включаем внутренние схемы, что приводит к необходимости составить более сложный запрос:

```
SELECT count(*) FROM pg_class
WHERE relkind = 'r'
AND notrelispartition
AND relnamespace NOT IN (
SELECT oid FROM pg_namespace
WHERE nspname IN ('information_schema','pg_catalog', 'pg_ toast')
AND nspname NOT LIKE 'pg_temp%' AND nspname NOT LIKE
'pg_ toast_temp%'
);
```

Для того чтобы посмотреть на размер реальных файлов, вам нужно убедиться, что вы включили каталог данных и все подкаталоги, а также все другие каталоги, содержащие табличные пространства.

Это может быть затруднительно, а также трудно выделить все разные части. Самый простой способ – задать в базе данных простой запрос, подобный этому:

```
SELECT pg_database_size(current_database());
```

Однако это ограничено только текущей базой данных. Если вы хотите узнать размер всех баз данных вместе взятых, вам потребуется запрос, например, следующий:

```
SELECT sum(pg_database_size(datname))frompg_database;
```

Максимальный поддерживаемый размер таблицы в конфигурации по умолчанию составляет 32 ТБ, и он не требует поддержки больших файлов операционной системой. Ограничения размера файловой системы не влияют на большие таблицы, поскольку они хранятся в нескольких файлах размером 1 ГБ.

При сканировании больших таблиц могут возникать проблемы с производительностью. Обновление индексов может занять гораздо больше времени, а производительность запросов может ухудшиться.

Для того чтобы увидеть размер таблицы воспользуйтесь командой:

```
cookbook=# selectpg_relation_size('pgbench_accounts')
```

Результат выполнения этой команды выглядит следующим образом:

```
pg_relation_size
------------------
13582336
(1 row)
```

Также можно увидеть общий размер таблицы, включая индексы и другие связанные пространства, следующим образом:

```
cookbook=# selectpg_total_relation_size('pgbench_accounts');
```

Результат выполнения этой команды выглядит следующим образом:

```
pg_total_relation_size
------------------------
```

```
15425536
(1 row)
```

Мы также можем использовать sql-команду, подобную этой

```
cookbook=# \dt+ pgbench_accounts
Listofrelations
Schema | Name | Type | Owner | Size |
Description
--------+-----------------+-------+--------+-------+---------
----
gianni | pgbench_accounts | table | gianni | 13 MB |
(1 row)
```

В PostgreSQL таблица состоит из множества отношений. Основное отношение – это таблица данных. Кроме того, существует множество дополнительных файлов данных. Каждый индекс, созданный в таблице, также является отношением. Большие значения данных помещаются во вторичную таблицу с именем TOAST, поэтому в большинстве случаев каждая таблица также имеет таблицу TOAST и индекс TOAST.

Каждое отношение состоит из нескольких файлов данных. Основные файлы данных разбиты на части по 1 ГБ. Первый файл не имеет суффикса, остальные имеют нумерованный суффикс (например, 2). Есть также файлы, помеченные _vmи _fsm, которые представляют собой карту видимости и карту свободного пространства соответственно. Они используются как часть операций по техническому обслуживанию. Они остаются довольно маленькими даже для очень больших таблиц.

Количество строк в таблице не ограничено, но таблица ограничена доступным дисковым пространством и памятью/пространством подкачки. Если вы сохраняете строки, размер которых превышает совокупный размер данных 2 КБ, то максимальное количество строк может быть ограничено 4 миллиардами или меньше.

Подсчет – это одно из самых простых операторов SQL, а также первый опыт многих людей с запросом PostgreSQL.

Из любого интерфейса команда SQL, используемая для подсчета строк команда, выглядит следующим образом:

```
SELECT count(*) FROM table;
```

Она вернет одно целое значение в качестве результата.

В psql, команда выглядит следующим образом:

```
cookbook=# selectcount(*) fromorders;
count
-------
345
(1row)
```

PostgreSQL может выбирать между двумя методами, доступными для вычисления через SQL функцию count(*). Оба доступны во всех текущих поддерживаемых версиях:

Первый называется последовательное сканирование. Мы получаем доступ ко всем блокам данных в таблице один за другим, считывание количества строк в каждом блоке. Если таблица находится на диске, это вызовет специальный шаблон доступа к диску, и инструкция будет достаточно быстрой.

Другой метод известен как сканирование только по индексу. Для этого требуется индексация таблиц, и он охватывает более общий случай, чем оптимизация SQL-запросов с помощью count(*).

3. КОНФИГУРАЦИЯ СЕРВЕРА

В этой главе мы рассмотрим следующие темы:

Планирование новой базы данных

Установка параметров конфигурации для сервера базы данных

Настройка параметров конфигурации в ваших программах

Поиск параметров конфигурации для вашего сеанса

Поиск параметров с нестандартными настройками

Настройка параметров для отдельных групп пользователей

Контрольный список базовой конфигурации сервера

3.1. ПЛАНИРОВАНИЕ НОВОЙ БАЗЫ ДАННЫХ

Планирование новой базы данных может оказаться непростой задачей. Ей легко увлечься, поэтому здесь мы представим некоторые идеи по планированию. Также легко полностью погрузиться в задачу, думая, что все, что вы знаете, вам когда-нибудь придется рассмотреть.

Напишите документ, который охватывает следующие пункты:

– проектирование базы данных – спланируйте дизайн своей базы данных;

– рассчитайте исходный размер базы данных.

– транзакционный анализ – как вы будете получать доступ к базе данных;

– посмотрите самые частые пути доступа (например, запросы).

– каковы требования к времени ответа;

– спецификация оборудования;

– первоначальные решения о производительности;

– выбрать операционную систему (операционные системы) и типы файловых систем;

– план локализации;

– определите кодировку сервера, местоположение и часовой пояс;

– план администрирования доступа и безопасности;

– определите клиентские системы и укажите необходимые драйверы;

– создайте роли согласно плану контроля доступа;

– укажите маршруты подключения и аутентификацию для сервера вpg_hba.conf.

– мониторинг – есть ли подключаемые модули PostgreSQL для решения по мониторингу, которое вы уже используете. Какие специфические для бизнеса метрики нам нужно отслеживать;

– план обслуживания – как будет поддерживаться работа во время обслуживания;

– план доступности – учитывайте требования к доступности.

Одна из наиболее важных причин для планирования вашей базы данных заблаговременно заключается в том, что модернизация некоторых элементов затруднена. Это особенно верно для серверной кодировки и локализации, которые могут привести к длительному простою и усилиям, если нам потребуется изменить их позже. Безопасность также намного сложнее настроить, когда система почти находится в рабочем состоянии.

3.2. УСТАНОВКА ПАРАМЕТРОВ КОНФИГУРАЦИИ ДЛЯ СЕРВЕРА БАЗЫ ДАННЫХ

Файл параметров – это основное место, которое используется для определения значений параметров сервера PostgreSQL. Все параметры можно задать в файле параметров, известном как postgresql.conf. Есть также два других файла параметров: pg_hba.confиpg_ident.conf. Оба они относятся к соединениям и безопасности. Bpg_settings контекст определяет, когда каждый параметр может быть установлен. В следующей таблице это классифицировано, чтобы мы могли видеть, какие действия необходимы для того, чтобы изменения вступили в силу. SET – это команда, но RELOAD и RESTART – это действия, а не конкретные команды. Некоторые параметры, отмеченные POSTMASTER, отмечены как исключения в следующей таблице. Эти параметры должны быть установлены на значение, меньшее или равное их значению в режиме ожидания. В результате, чтобы увеличить их на первичном узле, мы должны сначала увеличьте их на ВСЕХ резервных узлах, а затем перезапустить перед перезапуском основного узла – например, max_connections (рис. 3.1).

Если у вас есть прямой доступ к файлам параметров, то можно найти файл postgresql.conf. После изменения параметров необходимо отправить команду перезагрузки серверу, при этом PostgreSQL повторно прочитает файл postgresql.conf (и все другие файлы конфигурации). Существует несколько способов сделать это, в зависимости от вашего дистрибутива и операционной системы.

Context (Количество параметров этого типа в PGM, исключая любые добавленные расширения)	SET Устанавливается пользователем с помощью команды SET в сеансе. ALTER USER и(или) опцию подключения	RELOAD Изменено в конфигурационном файле: перезагрузка реализует изменения	RESTART Изменено в конфигурации: перезапуск основного узла для реализаций изменений	RESTARTALL Изменено в конфигурации: перезапуск основного и всех резервных узлов реализует изменение(я)
USER (133)	✓Только для локального сеанса	✓Для всех сеансов, которые еще не установлены	Не требуется, но будет применен при перезапуске	Неприменимый
SUPERUSER (43)	✓Но только суперпользователь	(как указано выше)	(как указано выше)	Неприменимый
SIGHUP (90)	X	✓	(как указано выше)	Неприменимый
POSTMASTER (54)	X	X	✓	Неприменимый
Exceptions: max_connect ions max_wal_senderstnax _worker_procsmax_pr epared_xactsmax_lock s_per_xactwal_levelw al_log_hints t rack_ccxnmit_t s	X	X	X	✓
INTERNAL (18)	X	X	X	X

Рис. 3.1. Контекст параметров показывает, когда изменения вступают в силу

Наиболее распространенный способ – выполнить следующую команду с тем же пользователем операционной системы, который запускал процесс сервера PostgreSQL:

pg_ctlreload

Эта команда сработает, если при установке использовался каталог данных по умолчанию; в противном случае необходимо указать правильный каталог данных с помощью опции -D.

4. ТАБЛИЦЫ И ДАННЫЕ

В этой главе рассматривается ряд общих вопросов, связанных с таблицами, работой с содержащимися в них данными.

Если поддерживать чистоту данных, то запросы будут выполняться быстрее и вызывать меньше ошибок приложений.

Первое, с чего надо начать – это выбрать хорошие имена для объектов базы данных.

Самый простой способ помочь другим людям понять базу данных – убедиться, что все объекты имеют осмысленное полное имя.

Потратьте некоторое время на то, чтобы поразмыслить над своей базой данных, чтобы убедиться, что у вас есть четкое представление о ее назначении и основных вариантах использования.

Вот моменты, которые следует учитывать при именовании объектов вашей базы данных:

– название соответствует существующим стандартам и практике;

– название ясно описывает роль или содержимое таблицы;

– для основных таблиц используйте короткие, выразительные названия;

– называйте таблицы, в которых будет производиться поиск данных приписывая название таблицы, с которой они связаны, например, account_status;

– для ассоциативных или связанных таблиц дописывайте имена основных таблиц, к которым они относятся, например, customer_account;

– убедитесь, что имя четко отличается от других похожих имен;

– будьте последовательны в сокращениях;

– используйте символы подчеркивания;

– не используйте прописной регистр такой как customer Account;

– избегайте имена, содержащие пробелы и точки с запятой;

– будьте последовательны при использовании множественного числа или не используйте вовсе;

– использует суффиксы для идентификации типа содержимого или домена объекта. PostgreSQL уже использует суффиксы для автоматически генерируемых объектов;

– не выбирайте имена, которые относятся к текущей роли или расположению объекта. Москва, потому что он существует на сервере в Москве. Этот сервер может быть перемещен, например, в Новосибирск;

– не выбирайте имена, которые подразумевают, что объект является единственным в своем роде, например, таблица с именем TEST или таблица с именем BACKUP_DATA. С другой стороны, такая информация может быть помещена в имя базы данных, которое обычно не используется внутри самой базы данных.

4.1. ВЫБОР ПОДХОДЯЩИХ ИМЕН ДЛЯ ОБЪЕКТОВ БАЗЫ ДАННЫХ

Избегайте использования акронимов вместо длинных имен таблиц. Например, money_allocation_decision намного лучше, чем MAD. Это особенно важно, так как PostgreSQL переводит имена в нижний регистр, так что это аббревиатура может быть непонятна.

Имя таблицы обычно используется как корень для других создаваемых объектов, поэтому не добавляйте суффиксы в названия или что-то подобное.

Имена объектов PostgreSQL могут содержать пробелы и символы смешанного регистра, если имена таблиц заключены в двойные кавычки. Это может вызвать некоторые трудности и проблемы с безопасностью.

Проблемы с чувствительностью к регистру часто могут быть проблемой для людей, которые больше привыкли работать с другими системами баз данных, такими как MySQL, или для людей, которые сталкиваются с проблемой переноса кода из MySQL.

Давайте создадим таблицу, в которой используется имя в кавычках со смешанными регистрами, например следующее:

```
CREATE TABLE "MyCust"
AS
SELECT * FROM mycust;
```

Если мы попытаемся получить доступ к этой таблице без использования регистра, мы получим эту ошибку:

```
postgres=# SELECT count(*) FROM mycust;
ERROR: relation "mycust" doesnotexist
LINE 1: SELECT * FROM mycust;
```

Тогда, напишем в правильном регистре:

```
postgres=# SELECT count(*) FROM MyCust;
ERROR: relation "mycust" doesnotexist
LINE 1: SELECT * FROM mycust;
```

Все равно не исполняется и, по сути, дает ту же самую ошибку.

Если вы хотите получить доступ к таблице, которая была создана с именами в кавычках, вы должны использовать имена в кавычках в запросе:

```
postgres=# SELECT count(*) FROM "MyCust";
```

Получим результат:

```
count
-------
5
(1 row)
```

Правило использования состоит в том, что если вы создаете свои таблицы, используя имена в кавычках, то вам нужно писать свой SQL, используя имена в кавычках. В качестве альтернативы, если ваш SQL использует имена в кавычках, вам, вероятно, придется создавать таблицы с использованием имен в кавычках.

Кроме того, PostgreSQL сворачивает все имена в нижний регистр при использовании в операторе SQL. Обратите внимание на эту команду:

```
SELECT * FROM mycust;
```

Это в точности то же самое, что и следующая команда:

```
SELECT * FROM MYCUST;
```

Это тоже самое, как и эта команда:

```
SELECT * FROM MyCust;
```

Однако это не то же самое, что следующая команда:

```
SELECT * FROM "MyCust";
```

Применение одного и того же имени и определения для столбцов.

Разумно спроектированные базы данных имеют плавные, простые для понимания определения. Это позволяет всем пользователям понимать значение данных в каждой таблице. Это важный способ устранения проблем с качеством данных.

Если вы хотите протестировать запросы из этой главы, используйте следующие примеры. В качестве альтернативы вы можете просто проверить наличие проблем в своей собственной базе данных:

```
CREATE SCHEMA s1;
CREATE SCHEMA s2;
CREATE TABLE s1.X (col1 smallint,col2 TEXT);
CREATE TABLE s2.X (col1 integer,col3 NUMERIC);
```

Сначала посмотрим, как идентифицировать столбцы, которые определены по-разному в разных таблицах, используя запрос к каталогу. Мы будем использовать запрос information_schema следующим образом:

```
SELECT table_schema
,table_name
,column_name
,data_type
||coalesce (' ' || text(character_maximum_length), ' ')
||coalesce (' ' || text(numeric_precision), ' ')
||coalesce(',' || text(numeric_scale), ' ')
asdata_type
FROM information_schema.columns
WHERE column_name IN
(SELECT
column_name
FROM
(SELECT
column_name
,data_type
,character_maximum_length
,numeric_precision
,numeric_scale
FROM information_schema.columns
```

```
WHERE table_schema NOT IN ('information_schema', 'pg_catalog')
GROUP BY
column_name
,data_type
,character_maximum_length
,numeric_precision ,numeric_scale
) derived
GROUP BY column_name
HAVING count(*) > 1
)
AND table_schema NOT IN ('information_schema', 'pg_catalog')
ORDER BY column_name ;
```

Запрос выдаст следующее:

```
table_schema | table_name | column_name | data_type
-------------- +------------ +------------- +---------------
s1 | x | col1 | smallint 16,0
s2 | x | col1 | integer 32,0
(2 rows)
```

Запрос на сравнение двух заданных таблиц является более сложным, поскольку существует множество способов, которыми таблицы могут быть похожими и в то же время немного разными. Следующий запрос ищет все таблицы с одинаковым именем (и, следовательно, разные схемы), которые имеют разные определения:

```
WITH table_definitionas
(SELECT table_schema
,table_name
,string_agg( column_name || ' ' || data_type
        , ',' ORDER BY column_name
        ) AS def
FROM information_schema.columns
WHERE table_schema NOT IN ( 'information_schema' , 'pg_catalog')
GROUP BY table_schema , table_name
)
, unique_definitionas
```

```
(SELECT DISTINCT table_name
,def
FROM table_definition
)
,multiple_definitionas
(SELECT table_name
FROM unique_definition
GROUP BY table_name
HAVING count( * ) > 1
)
SELECT table_schema
,table_name
,column_name
,data_type
FROM information_schema.columns
WHERE table_name
    IN ( SELECT table_name
        FROM multiple_definition )
ORDER BY table_name
,table_schema
,column_name
;
```

Результат запроса:

```
table_schema | table_name | column_name | data_type
-------------- +------------ +------------- +-----------
s1 | x | col1 | smallint
s1 | x | col2 | text
s2 | x | col1 | integer
s2 | x | col3 | numeric
(4 rows)
```

Описания таблиц хранятся в PostgreSQL, и к ним можно получить доступ с помощью каталога информационных схем. Могут быть веские причины, по которым определения различаются. В приведенных запросах были исключены собственные внутренние таблицы PostgreSQL, потому что между двумя

каталогами есть похожие имена: реализация Postgresql стандартной информационной схемы SQL и собственная внутренняя схема pg_catalogPostgreSQL. Эти запросы довольно сложны. На самом деле, существует еще большая сложность, которую мы можем добавить к этим запросам для сравнения всевозможных вещей, таких как значения по умолчанию или ограничения.

4.2. ВЫЯВЛЕНИЕ И УДАЛЕНИЕ ДУБЛИКАТОВ

Реляционные базы данных основаны на идее, что элементы данных могут быть однозначно идентифицированы. Как бы мы ни старались, всегда откуда-то будут поступать недостоверные данные. В этих рецептах показано, как диагностировать этот беспорядок и устранять его.

Давайте начнем с рассмотрения примера таблицы cust. Он имеет повторяющееся значение в customerid:

```
CREATE TABLE cust (
customerid BIGINT NOT NULL
,firstname TEXT NOT NULL
,lastname TEXT NOT NULL
,age INTEGER NOT NULL);
INSERT INTO cust VALUES (1, 'Philip', 'Marlowe', 33);
INSERT INTO cust VALUES (2, 'Richard', 'Hannay', 37);
INSERT INTO cust VALUES (3, 'Harry', 'Palmer', 36);
INSERT INTO cust VALUES (4, 'Rick', 'Deckard', 4);
INSERT INTO cust VALUES (4, 'Roy', 'Batty', 41);
postgres=# SELECT * FROM cust ORDER BY 1;
customerid | firstname | lastname | age
------------ +----------- +---------- +-----
1 | Philip | Marlowe | 33
2 | Richard | Hannay | 37
3 | Harry | Palmer | 36
4 | Rick | Deckard | 4
4 | Roy | Batty | 41
(5 rows)
```

Прежде чем удалять дублирующиеся данные, помните, что иногда неверны не сами данные, а ваше понимание их. В таких случаях может оказаться, что вы неправильно нормализовали свою модель базы данных и что вам необходимо включить дополнительные таблицы для учета формы данных. Вы также можете обнаружить, что повторяющиеся строки вызваны вашим решением исключить столбец где-то ранее в процессе загрузки данных. Проверь дважды, отрежь один раз.

Далее, во-первых, определите дубликаты с помощью запроса, например, следующего:

```
CREATE UNLOGGED TABLE
dup_cust AS
SELECT *
FROM cust
WHERE customerid IN
    (SELECT customerid
    FROM cust
    GROUP BY customerid
    HAVING count(*) > 1);
```

UNLOGGED таблица может быть создана с меньшими затратами ввода-вывода, поскольку в нее не записывается WAL. Это лучше, чем временная таблица, потому что она не исчезнет, если вы отключитесь, а затем снова подключитесь. Другая сторона медали заключается в том, что вы теряете ее содержимое после сбоя, но это не так уж плохо, потому что если вы решите использовать незарегистрированную таблицу, то вы сообщаете PostgreSQL, что вы сможете воссоздать содержимое этой таблицы в (маловероятном) случае сбоя. Результаты могут быть использованы для определения неверных данных вручную, и вы можете устранить проблему, выполнив следующие действия.

1. Объедините две строки, чтобы получить наилучшее представление данных, если это необходимо. Это может использовать значения из одной строки для обновления строки, которую вы решите сохранить, как показано здесь:

```
UPDATE cust
SET age = 41
WHERE customerid = 4 AND lastname = 'Deckard';
```

2. Удалите оставшиеся ненужные строки:

```
DELETE FROM cust
WHERE customerid = 4 AND lastname = 'Batty';
```

В некоторых случаях строки данных могут быть полностью идентичными, создадим пример:

```
CREATE TABLE new_cust (customerid BIGINT NOT NULL);
INSERT INTO new_cust VALUES (1), (1), (2), (3), (4), (4);
```

Таблица new_cust выглядит следующим образом:

```
postgres=# SELECT * FROM new_cust ORDER BY 1;
customerid
------------
         1
         2
         3
         4
         4
(5 rows)
```

В отличие от предыдущего случая, мы не можем отличить данные по частям вообще, поэтому мы не можем удалить повторяющиеся строки без какого-либо ручного процесса. SQL – это язык, основанный на наборах, поэтому выбрать только одну строку из набора немного сложнее, чем хотелось бы большинству людей.

В этих обстоятельствах мы должны использовать несколько иную процедуру для обнаружения дубликатов. Мы будем использовать скрытый столбец с именем ctid. Он обозначает физическое местоположение строки, которую вы наблюдаете – все повторяющиеся строки будут иметь разные значения ctid. Шаги заключаются в следующем:

1. Во-первых, запускаем транзакцию:

```
BEGIN;
```

2. Затем мы заблокируем таблицу, чтобы предотвратить любые операции INSERT, UPDATE или DELETE, которые изменят список дубликатов и(или) изменят их значение ctid:

```
LOCK TABLE new_cust IN SHARE ROW EXCLUSIVE MODE;
```

3. Теперь найдите все дубликаты, отслеживая минимальное значение ctid, что удалять его:

```
CREATE TEMPORARY TABLE dups_cust AS
SELECT customerid, min(ctid) AS min_ctid
FROM new_cust
GROUP BY customerid
HAVING count(*) > 1;
```

4. Затем мы можем удалить каждый дубликат, за исключением дубликата с минимальным ctid:

```
DELETE FROM new_cust
USING dups_cust
WHERE new_cust.customerid = dups_cust.customerid
AND new_cust.ctid != dups_cust.min_ctid;
```

5. Мы завершаем транзакцию, которая также снимает блокировку, которую мы наложили ранее:

```
COMMIT;
```

6. Наконец, очищаем таблицу после удалений:

```
VACUUM new_cust;
```

Первый запрос группирует строки в уникальном столбце и подсчитывает количество строк. Все, что содержит более одной строки, должно быть вызвано повторяющимися значениями. Если мы ищем дубликаты более чем одного столбца (или даже всех столбцов), то мы должны использовать SQL-запрос следующей формы:

```
SELECT *
FROM mytable
WHERE (col1, col2, ..., colN) IN
    (SELECT col1, col2, ..., colN
    FROM mytable
```

```
GROUP BY col1, col2, ... , colN
HAVING count(*) > 1);
```

Здесь, col1, col2, и так далее colN – столбцы ключа.

Обратите внимание, что для этого типа запроса может потребоваться отсортировать полную таблицу по всем ключевым столбцам. Для этого потребуется пространство сортировки, равное размеру таблицы, поэтому вам лучше сначала подумать, прежде чем запускать этот SQL для очень больших таблиц. Вероятно, вам пригодится большой параметр work_mem для этого запроса, вероятно, 128 МБ или больше.

Запрос DELETE FROM ... USING, который используется, работает только с PostgreSQL, потому что он использует значение ctid, которое является внутренним идентификатором каждой строки в таблице. Если вы хотите выполнить этот запрос более чем к одному столбцу, как мы делали ранее в этой главе, вам нужно будет расширить запросы на шаге 3 следующим образом:

```
SELECT customerid, customer_name, ..., min(ctid) AS min_ctid
FROM ...
GROUP BY customerid, customer_name, ... ...;
```

Затем расширьте запрос на шаге 4, например:

```
DELETE FROM new_cust
...
WHERE new_cust.customerid = dups_cust.customerid
AND new_cust.customer_name = dups_cust.customer_name
AND ...
AND new_cust.ctid != dups_cust.min_ctid;
```

Предыдущий запрос работает путем группировки всех строк с одинаковыми значениями, а затем поиска строки с наименьшим значением ctid. Самый минимальный будет ближе к началу таблицы, поэтому дубликаты будут удалены из дальнего конца таблицы. Когда мы запускаем VACUUM, мы можем обнаружить, что таблица становится меньше, потому что мы удалили строки с дальнего конца.

Команды BEGIN и COMMIT объединяют команды LOCK и DELETE в единую транзакцию, которая является обязательной. В противном случае блокировка будет снята сразу же после исполнения. Еще одна причина использовать единую транзакцию заключается в том, что мы всегда можем выполнить

откат, если что-то пойдет не так, что хорошо, когда мы удаляем данные из текущей таблицы.

Предотвращение дублирования – один из самых важных аспектов качества данных для любой базы данных. PostgreSQL предлагает некоторые полезные функции в этой области, выходящие за рамки большинства реляционных баз данных.

Определите набор столбцов, который вы хотите сделать уникальным. Это относится ко всем строкам или только к подмножеству строк?

Начнем с примера:

```
postgres=# SELECT * FROM new_cust;
customerid
------------
          1
          2
          3
          4
(4 rows)
```

Чтобы предотвратить дублирование строк, нам нужно создать уникальный индекс, который сервер базы данных может использовать для обеспечения уникальности определенного набора столбцов. Мы можем сделать это тремя способами для основных типов данных.

1. Создать ограничение по первичному ключу для набора столбцов. Допускается только один ключ в таблице. Значения строк данных не должны быть NULL, так как мы принудительно назначаем столбцам NOTNULL, если они еще не являются таковыми:

```
ALTER TABLE new_cust ADD PRIMARY KEY(customerid);
```

Запрос создает новый индекс с именем new_cust_pkey.

2. Создать ограничение уникальности на набор столбцов. Мы можем использовать их вместо/или с первичным ключом. Количество таких столбцов в таблице не ограничено. Допустимы значения NULL в столбцах:

```
ALTER TABLE new_cust ADD UNIQUE(customerid);
```

Запрос создает новый индекс с именем new_cust_customerid_key.

3. Создайте уникальный индекс для набора столбцов:

```
CREATE UNIQUE INDEX ON new_cust (customerid);
```

Запрос создает новый индекс с именем new_cust_customerid_idx.

Все эти методы исключают дубликаты, просто с немного разным синтаксисом. Все они создают индекс, но только первые два создают формальное ограничение. Каждый из этих методов можно использовать, когда у нас есть первичный ключ или уникальное ограничение, использующее несколько столбцов.

Последний метод важен, поскольку он позволяет вам указать предложение WHERE в индексе. Это может быть полезно, если вы знаете, что значения столбцов уникальны только при определенных обстоятельствах. Результирующий индекс называют частичный индекс (partialindex)

Предположим, наши данные выглядели так:

```
postgres=# SELECT * FROM partial_unique;
```

Запрос дает следующий результат:

```
customerid | status| close_date
-----------+--------+------------
        1 | OPEN |
        2 | OPEN |
        3 | OPEN |
        3 | CLOSED | 2010-03-22
(4 rows)
```

Затем мы можем поместить частичный индекс в таблицу, чтобы обеспечить уникальность customerid только для status = 'OPEN', например:

```
CREATE UNIQUE INDEX ON partial_unique (customerid)
        WHERE status = 'OPEN';
```

Если необходимо применить ограничение уникальности для более сложных типов данных, может понадобиться использовать расширенный синтаксис. Несколько примеров показаны далее.

Давайте начнем с простого примера: создайте таблицу из блоков и поместите в нее образцы данных. Возможно, вы впервые видите синтаксис типов данных PostgreSQL, так что обратите на это внимание:

```
postgres=# CREATE TABLE boxes (nametext, positionbox);
```

```
CREATE TABLE
postgres=# INSERT INTO boxes VALUES ('First', box '((0,0), (1,1))');
INSERT 0 1
postgres=#
INSERT INTO boxes VALUES ('Second', box '((2,0), (2,1))');
INSERT 0 1
postgres=# SELECT * FROM boxes;
name | position
--------+-------------
First| (1,1),(0,0)
Second | (2,1),(2,0)
(2 rows)
```

Мы видим два прямоугольника, которые не соприкасаются и не перекрываются, основываясь на их *x* и *y* координатах.

Чтобы усилить здесь уникальность, мы хотим создать ограничение, которое отбрасывает любые попытки добавить позицию, перекрывающуюся с любым существующим блоком. Оператор перекрытия для типа данных блока определяется как &&, поэтому мы используем следующий синтаксис, чтобы добавить ограничение:

```
ALTER TABLE boxes ADD EXCLUDE USING gist (position WITH &&);
```

Запрос создает новый индекс с именем new_cust_customerid_excl, а дубликаты исключаются:

```
# insertintonew_cust VALUES (4);
ERROR: conflictingkeyvalueviolatesexclusionconstraint
"new_cust_customerid_excl"
DETAIL: Key (customerid)=(4) conflictswithexistingkey (customerid)=(4).
```

Возможно использовать тот же синтаксис с базовыми типами данных. Итак, четвертый способ выполнения нашего первого примера будет следующим:

```
ALTER TABLE new_cust ADD EXCLUDE (customerid WITH =);
```

Запрос создает новый индекс с именем new_cust_customerid_excl, а дубликаты исключаются:

```
# insertintonew_cust VALUES (4);
```

```
ERROR: conflictingkeyvalueviolatesexclusionconstraint
"new_cust_customerid_excl" DETAIL: Key (customerid)=(4)
conflictswithexistingkey (customerid)=(4).
```

Уникальность всегда обеспечивается индексом.

Каждый индекс определяется с помощью оператора типа данных. Когда вставляется новая строка или обновляется набор значений столбца, мы используем оператор для поиска существующих значений, конфликтующих с новыми данными.

Таким образом, чтобы обеспечить уникальность, нам нужен индекс и оператор поиска, определенные для типов данных столбцов. Когда определяем обычные ограничения UNIQE, мы просто предполагаем, что имеем в виду оператор равенства (=) для типа данных. Синтаксис EXCLUDE предлагает более богатый синтаксис, позволяющий нам выразить одну и ту же проблему с различными типами данных и операторами.

Возможно иметь уникальность в наборе столбцов без создания индекса. Это может быть полезно, если необходимо обеспечить уникальность, но не разрешать поиск по индексу.

Для этого возможно сделать одно из:

- использовать серийный тип данных;
- вручную изменить значение по умолчанию на функцию последовательности nextval().

Каждое из них будет предоставлять уникальное значение для использования в качестве ключа строки. Уникальность не применяется принудительно, и при этом не будет определено уникальное ограничение. Таким образом, все еще существует вероятность того, что кто-то может сбросить последовательность до более раннего значения, что в конечном итоге приведет к дублированию значений.

Учтите также, что этот метод предоставляет уникальное значение по умолчанию, которое не используется, когда пользователь указывает явное значение, например:

```
CREATE TABLE t(idserial, descrtext);
INSERT INTO t(descr) VALUES ('Firstvalue');
INSERT INTO t(id,descr) VALUES (1,'Cheating!');
```

Наконец, возможно создавать в основном уникальные данные, например, использовать функцию clock_timestamp() для определения времени возрастания с разрешением в микросекунду.

Пример из реального мира – IP-адрес диапазон распределения.

Проблема заключается в назначении диапазонов IP-адресов и одновременном обеспечении того, чтобы мы не выделяли (или потенциально не выделяли) одни и те же адреса разным людям или целям. Это легко сделать, если мы отслеживаем каждый отдельный IP-адрес, но сделать это гораздо сложнее, если мы хотим работать только с диапазонами IP-адресов.

Первоначально возможно подумать о разработке базы данных следующим образом:

```
CREATE TABLE iprange (iprange_startinet, iprange_stopinet, ownertext);
INSERT INTO iprange VALUES ('192.168.0.1','192.168.0.16', 'Simon');
INSERT INTO iprange VALUES ('192.168.0.17','192.168.0.24', 'Gianni');
INSERT INTO iprange VALUES ('192.168.0.32','192.168.0.64', 'Gabriele');
```

Однако, если разобраться, то невозможно создать уникальное ограничение, которое применяет ограничение на модель, чтобы избежать перекрывающихся диапазонов. Возможно создать последующий триггер, который проверяет существующие значения, но это будет не элегантно.

PostgreSQL предлагает лучшее решение, основанное на типах диапазонов. Фактически, каждый тип данных, который поддерживает класс оператора btree (т.е. способ упорядочивания любых двух заданных значений), может быть использован для создания типа диапазона. В нашем случае SQL выглядит следующим образом:

```
CREATE TYPE inetrange AS RANGE (SUBTYPE = inet);
```

Эта команда создает новый тип данных, который может представлять диапазоны значений inet, т.е. IP-адресов. Теперь мы можем использовать этот новый тип при создании таблицы:

```
CREATE TABLE iprange2 (iprangeinetrange, ownertext);
```

Эта новая таблица может быть заполнена как обычно. Нам просто нужно сгруппировать крайние значения каждого диапазона в одно значение, как показано ниже:

```
INSERT INTO iprange2 VALUES ('[192.168.0.1,192.168.0.16]', 'Simon');
INSERT INTO iprange2 VALUES ('[192.168.0.17,192.168.0.24]', 'Gianni');
INSERT INTO iprange2 VALUES ('[192.168.0.32,192.168.0.64]', 'Gabriele');
```

Теперь мы можем создать уникальное ограничение исключения для таблицы, используя следующий синтаксис:

```
ALTER TABLE iprange2
ADD EXCLUDE USING GIST (iprange WITH &&);
```

Если мы попытаемся вставить диапазон, который перекрывается с любым из существующих диапазонов, PostgreSQL остановит работу:

```
INSERT INTO iprange2
VALUES ('[192.168.0.10,192.168.0.20]', 'Somebodyelse');
ERROR: conflictingkeyvalueviolatesexclusionconstraint
"iprange2_iprange_excl"
DETAIL: Key (iprange)=([192.168.0.10,192.168.0.20]) conflicts
withexistingkey (iprange)=([192.168.0.1,192.168.0.16]).
```

Пример из реального мира – диапазон времени

Во многих базах данных будут таблицы исторических данных с данными, которые имеют значение START_DATE и значение END_DATE или что-то подобное. Как и в предыдущем примере, возможно элегантно решить эту проблему с помощью типа range. На самом деле, этот пример еще проще – нам не нужно создавать тип диапазона, поскольку наиболее распространенные случаи уже встроены – т.е. целые числа, десятичные значения, даты и временные метки с часовым поясом и без него.

4.3. ПОИСК УНИКАЛЬНОГО КЛЮЧА ДЛЯ НАБОРА ДАННЫХ

Иногда может быть трудно найти уникальный набор ключевых столбцов, описывающих данные. В этом примере проанализируем данные в базе данных, что позволит нам идентифицировать столбец (столбцы), которые вместе образуют уникальный ключ. Это полезно, когда некоторые ключи не задокументированы, не определены или определены неправильно.

Начнем с небольшой таблицы, где ответ достаточно очевиден:

```
postgres=# select * from ord;
```

Результат выглядит следующим образом:

```
orderid | customerid | amt
---------+------------+--------
10677 | 2 | 5.50
5019 | 3 | 277.44
9748 | 3 | 77.17
(3 rows)
```

Нет необходимости делать это методом прямого анализа. Проверка всех значений столбцов, чтобы определить, какая из них уникальна, может занять много времени и не исключает ошибки.

Давайте воспользуемся собственным оптимизатором статистики PostgreSQL. Запустите следующую команду в таблице, чтобы получить пример статистики:

```
postgres=# analyze ord;
ANALYZE
```

Этот запрос работает быстро и не заставит себя долго ждать. Теперь можно изучить соответствующий столбец статистики:

```
postgres=# SELECT attname, n_distinct
                  FROM pg_stats
                  WHERE schemaname = 'public'
                  AND tablename = 'ord';
attname | n_distinct
------------ +------------
orderid | -1
customerid| -0.666667
amt | -1
(3 rows)
```

Предыдущий пример был выбран потому, что у нас есть два потенциальных ответа. Если значение n_distinct равно –1, то столбец считается уникальным в пределах исследуемой выборки строк.

Затем нужно провести анализ, чтобы решить, являются ли один или оба из этих столбцов уникальными случайно или как часть дизайна базы данных, которая их создала.

Возможно, что нет ни одного столбца, однозначно идентифицирующего строки. Наличие нескольких ключевых столбцов довольно распространенное явление. Если ни один из столбцов не был уникальным, то нам следует начать поиск уникальных ключей, которые представляют собой комбинации наиболее уникальных столбцов. Следующий запрос показывает частотное распределение для таблицы, где значение встречается дважды в одном случае, а другое значение встречается только один раз:

```
postgres=# SELECT num_of_values, count(*)
                FROM (SELECT customerid, count(*) AS num_of_values
                          FROM ord
                          GROUP BY customerid) s
                GROUP BY num_of_values
                ORDER BY count(*);
num_of_values | count
---------------+-------
2 | 1
1 | 1
(2 rows)
```

Мы можем изменить запрос, чтобы он включал несколько столбцов, например:

```
SELECT num_of_values, count(*)
FROM (SELECT customerid, orderid, amt
            ,count(*) AS num_of_values
            FROM ord
            GROUP BY customerid, orderid, amt
            ) s
GROUP BY num_of_values
ORDER BY count(*);
```

Когда мы найдем уникальный набор столбцов, результатом этого запроса будет только одна строка, как показано в следующем примере:

```
num_of_values | count
-----------------+-------
1 | 3
```

По мере того, как мы будем приближаться к поиску ключа, мы увидим, что распределение становится все плотнее и плотнее. Итак, процедура выглядит следующим образом.

1. Выберите для начала один столбец.
2. Вычислите соответствующее частотное распределение.
3. Если в результате получается несколько строк, добавьте еще один столбец и повторите с шага 2. В противном случае это означает, что вы нашли набор столбцов, удовлетворяющих ограничению уникальности.

Теперь вы должны убедиться, что набор столбцов минимален – например, проверить, возможно ли удалить один или несколько столбцов, не нарушая ограничения уникальности.

1. Протестируйте каждый столбец, рассчитав частотное распределение на всех столбцах.
2. Если частотное распределение состоит из одной строки, то столбец не нужен для ограничения уникальности. Удалите его из набора столбцов и повторите, начиная с шага 1.

В противном случае вы нашли минимальный набор столбцов, который можно назвать ключом для этой таблицы.

Поиск уникального ключа возможен для программы, но в большинстве случаев человек может сделать это намного быстрее, просматривая такие вещи, как имена столбцов, внешние ключи или бизнес-процессы.

Команда ANALYZE работает, беря выборку табличных данных и затем выполняя статистический анализ результатов. Значение n_distinct имеет два разных значения, в зависимости от его знака: если оно положительное, то это оценка количества различных значений для столбца; если отрицательное, то это оценка плотности таких различных значений. Например, n_distinct = –0,2 означает, что ожидается, что таблица из 1 миллиона строк будет содержать 200 000 различных значений, в то время как n_distinct = 5 означает, что мы ожидаем всего 5 различных значений.

4.4. ГЕНЕРАЦИЯ ТЕСТОВЫХ ДАННЫХ

Администраторам баз данных часто необходимо генерировать тестовые данные по разным причинам, будь то для настройки тестовой базы данных или просто для создания тестового случая для проблемы с производительностью SQL.

Для создания таблицы тестовых данных нам потребуется следующее:

– несколько строк;

– несколько столбцов;

– какое-то упорядочивание.

Шаги следующие:

1. Во-первых, сгенерируйте несколько строк данных. Можно использовать способ через функции set-returning. Можно написать свои собственные данные, хотя проще воспользоваться полезными функциями PostgreSQL.

2. Сгенерируем последовательность строк, используя запрос, подобный следующему:

```
postgres=# SELECT * FROM generate_series(1,5);
generate_series
-----------------
               1
               2
               3
               4
               5
(5 rows)
```

3. Кроме того, можно создать список дат:

```
postgres=# SELECT date(t)
FROM generate_series(now(),
now() + '1 week', '1 day') AS f(t);
date
------------
2021-08-25
2021-08-26
2021-08-27
```

2021-08-28
2021-08-29
2021-08-30
2021-08-31
2021-09-01
(8 rows)

4. Затем мы хотим сгенерировать значение для каждого столбца в таблице test. Мы можем разбить действие на ряд шагов, используя следующие примеры в качестве руководства:

– любая из этих функций может быть использована для генерации как строк, так и разумных значений первичного ключа для них;

– для случайного значения integer это функция:

```
(random()*(2*10^9))::integer
```

– для случайного значения bigint это функция:

```
(random()*(9*10^18))::bigint
```

– для случайного значения numeric это функция:

```
(random()*100.)::numeric(5,2)
```

– для строки произвольной длины вплоть до максимальной длины это функция:

```
repeat('1',(random()*40)::integer)
```

– для подстроки произвольной длины функция выглядит следующим образом:

```
substr('abcdefghijklmnopqrstuvwxyz',1, (random()*25)::integer)
```

– вот функция для случайной строки из списка строк:

```
(ARRAY['one','two','three'])[0.5+random()*3]
```

6. Наконец, мы можем объединить способы для создания нашей таблицы:

```
postgres=# SELECT key
                              ,(random()*100.)::numeric(4,2)
,repeat('1',(random()*25)::integer)
```

```
FROM generate_series(1,10) AS f(key);
Key | numeric | repeat
------+----------+------------------------
1 | 83.05 | 1111
2 | 5.28 | 11111111111111
3 | 41.85 | 1111111111111111111111
4 | 41.70 | 11111111111111111
5 | 53.31 | 1
6 | 10.09 | 1111111111111111
7 | 68.08 | 111
8 | 19.42 | 1111111111111111
9 | 87.03 | 11111111111111111111
10 | 70.64 | 111111111111111
(10 rows)
```

7. В качестве альтернативы мы можем использовать случайный порядок:

```
postgres=# SELECT key
                    ,(random()*100.)::numeric(4,2)
,repeat('1',(random()*25)::integer)
                    FROM generate_series(1,10) AS f(key)
                    ORDERBYrandom() * 1.0;
key| numeric | repeat
-----+-----------+-------------------------
4 | 86.09 | 1111
10 | 28.30 | 11111111
2 | 64.09 | 111111
8 | 91.59 | 111111111111111
5 | 64.05 | 11111111
3 | 75.22 | 11111111111111111
6 | 39.02 | 1111
7 | 20.43 | 1111111
1 | 42.91 | 11111111111111111111
9| 88.64 | 1111111111111111111111
(10 rows)
```

Функции, возвращающие множество значений, буквально возвращают набор строк. Это позволяет использовать их либо в предложении FROM, как если бы они были таблицей, либо в предложении SELECT. Набор функций generate_series() возвращает либо даты, либо целые числа, в зависимости от типов данных используемых для входных параметров.

Оператор: используется для преобразования между типами данных. В примере для случайной строки из списка строк используются массивы PostgreSQL. Вы можете создать массив, используя синтаксическую конструкцию ARRAY, а затем использовать целое число для ссылки на один из элементов в массиве. В нашем случае мы использовали случайный индекс.

Случайная выборка данных.

Администраторов баз данных могут попросить настроить сервер и заполнить его тестовыми данными. Часто встречается ситуация, когда на сервере используется старое оборудование, возможно, с дисками небольшого размера. В этой ситуации выборка данных из общего массива данных имеет важное значение.

Целью выборки является уменьшение размера набора данных и повышение скорости последующего анализа. Некоторые виды статистики настолько часто используют с выборкой, что даже не возникает сомнений, допустимо ли использование выборки данных или это может вызвать какие-то осложнения или ошибки.

Стандартный способ выполнения выборки в SQL заключается в добавлении предложения TABLESAMPLE к оператору SELECT.

Возьмем случайную выборку предложенного набора данных (например, таблицу данных). Нужно понимать, что не существует простого инструмента для вычленения выборки из базы данных. Было бы здорово, если бы они были, но их нет.

1. Сначала мы рассмотрим использование SQL для получения выборки. Для этого необходимо использовать команду TABLESAMPLE. Посмотрите следующий пример:

```
postgres=# SELECT count(*) FROM mybigtable;
count
-------
10000
(1 row) postgres=# SELECT count(*) FROM mybigtable
```

```
                                        TABLESAMPLE BERNOULLI(1);
count
-------
106
(1 row)
postgres=# SELECT count(*) FROM mybigtable
                                        TABLESAMPLE BERNOULLI(1);
count
-------
99
(1 row)
```

2. Здесь предложение TABLESAMPLE применяется к большой таблице, SELECT указывает, что производится случайная выборка, в то время как ключевое слово BERNOULLI обозначает используемый метод выборки, а цифра 1 в круглых скобках указывает процент строк, которые необходимо учитывать в выборке, т.е. 1%.

3. Теперь нам нужно извлечь выборочные данные из базы данных, что сложно по нескольким причинам. Во-первых, нет возможности указать предложение WHERE для pg_dump. Во-вторых, если создать запрос, содержащий предложение WHERE, pg_dump выведет только свойства запроса, а не результат запроса.

4. Можно использовать pg_dump для создания дампа всей базы данных, кроме набора таблиц, можно создать выборочный дамп, подобный этому:

```
pg_dump --exclude-table=mybigtable > db.dmp
pg_dump --table=mybigtable --schema-only > mybigtable.
schema
psql -c '\copy (SELECT * FROM mybigtable
                                  TABLESAMPLE BERNOULLI (1)) to
mybigtable.dat'
```

Затем выполнить перезагрузку дампа в отдельную базу данных с помощью следующих команд:

```
psql -f db.dmp
psql -f mybigtable.schema
psql -c '\copy mybigtable from mybigtable.dat'
```

В целом, совет – использовать выборку с осторожностью. В общем, лучше всего применять ее только к нескольким очень большим таблицам, учитывая как математические проблемы, связанные с планом выборки, так и сложность извлечения данных.

Механизм извлечения демонстрирует возможности инструментов командной строки psql и pg_dump PostgreSQL, поскольку pg_dump позволяет вам включать или исключать объекты и выводить всю таблицу (или только ее схему), в то время как psql позволяет вам выводить результат произвольного запроса в файл.

Предложение BERNOULLI определяет метод выборки – т.е. PostgreSQL берет случайную выборку, выполняя полное сканирование таблицы, а затем выбирая каждую строку с требуемой вероятностью (здесь 1%).

Другим встроенным методом выборки является SYSTEM, который считывает случайную выборку страниц таблицы, а затем включает все строки на этих страницах; как правило, это быстрее, учитывая, что выборки обычно немного меньше оригинала, но на случайность выбора влияет то, как физически расположены строки на диске, что делает его пригодным только в некоторых случаях.

Вот пример, который показывает, в чем проблема. Предположим, вы берете словарь, вырываете несколько страниц, а затем выделяете в них все слова; вы получите случайную выборку, состоящую из нескольких кластеров последовательных слов. Этого достаточно, если вы хотите оценить среднюю длину слова, но не для анализа среднего количества слов для каждой начальной буквы. Причина в том, что начальная буква слова сильно коррелирует с расположением слов на страницах, а длина слова – нет.

Здесь не обсуждалось, насколько случайным является предложение TABLESAMPLE, однако, достаточно просто расширить PostgreSQL дополнительными функциями или методами выборки, поэтому, если необходим другой механизм выборки, можно найти внешний генератор случайных чисел и создать новый метод выборки для предложения TABLESAMPLE. PostgreSQL включает в себя два дополнительных метода выборки, tsm_system_rows и tsm_system_time, для расширения начальных возможностей.

Метод tsm_system_rows не работает с процентами; вместо этого числовой аргумент интерпретируется как количество возвращаемых строк. Точно также

метод tsm_system_time будет рассматривать аргумент как число миллисекунд, затраченное на получение случайной выборки.

Эти два метода включают слово system в свое название, поскольку они используют выборку на уровне блоков, то на их случайность влияет ограничение кластеризации.

Простая случайная выборка может сделать конечную выборку смещенной в сторону более часто встречающихся данных. Например, если у вас есть 1% выборки данных, в которой некоторые виды данных встречаются только 0,001% времени, вы можете получить набор данных, в котором нет каких-либо выпадающих данных.

Что вы можете сделать, так это предварительно кластеризовать ваши данные и взять разные образцы из каждой группы, чтобы убедиться, что у вас есть выборочный набор данных, который включает в себя гораздо больше исходных атрибутов. В качестве простого метода можно использовать следующее:

Включить 1% нормальных данных.

Включить 25% внешних данных.

4.5. ЗАГРУЗКА ДАННЫХ ИЗ ЭЛЕКТРОННОЙ ТАБЛИЦЫ

Электронные таблицы – наиболее очевидный метод хранения данных. Исследования, проведенные в ряде компаний по всему миру, неизменно показывают, что более 50% информации этих компаний хранятся в виде электронных таблицах или небольших файловых базах данных. Загрузка данных из этих источников является частой и важной задачей для многих администраторов баз данных.

Электронные таблицы объединяют данные, презентации и программы в один файл. Это идеально подходит для опытных пользователей, желающих работать продуктивно. Как и в случае с другими реляционными базами данных, PostgreSQL в основном работает с самыми низкими уровнями данных, поэтому извлечение данных из этих электронных таблиц может представлять некоторые проблемы.

Можем легко обрабатывать данные электронной таблицы, если макет этой электронной таблицы соответствует определенной форме, а именно:

– каждый столбец электронной таблицы является одним столбцом одной таблицы.

– каждая строка электронной таблицы является одной строкой одной таблицы.

– данные находятся на одном рабочем листе электронной таблицы.

– при желании в первой строке отображается список описаний/заголовков столбцов.

Это очень простой макет, и чаще всего в электронной таблице будут присутствовать другие элементы, такие как заголовки, комментарии, константы для использования в формулах, сводные строки, макросы и изображения. Если вы находитесь в таком положении, лучшее, что можно сделать, – это создать новый лист внутри электронной таблицы, создать структуру таблицы в виде, описанном ранее, а затем настроить ссылки между листами для ввода данных. Примером перекрестной ссылки на рабочий лист может быть =Sheet2.A1. Вам понадобится отдельный рабочий лист для каждого набора данных, который станет одной таблицей в PostgreSQL. Однако вы можете загрузить несколько листов в одну таблицу.

Рассмотрим пример, где данные электронной таблицы загружаются в базу данных:

1. Если данные электронной таблицы аккуратно размещены на одном листе, как показано на следующем снимке экрана, вы можете перейти к Файл|Сохранить как, а затем выбрать CSV в качестве типа сохраняемого файла:

2. Это позволит экспортировать текущий рабочий лист в файл следующим образом:

```
"Key","Value"
1,"c"
2,"d"
```

Рис. 5.1. Пример очень простой электронной таблицы

3. Затем создадим таблицу для загрузки данных, используя команду psql:

```
CREATE TABLE example
(key integer
,value text);
```

4. Затем загружаем его в таблицу PostgreSQL, используя следующие команды psql:

```
postgres=# \COPY sample FROM sample.csv CSV HEADER
postgres=# SELECT * FROM sample;
key | value
-----+-------
1 | c
2 | d
```

5. Альтернативная команда для командной строки выглядит следующим образом:

```
psql -c '\COPY sample FROM sample.csv CSV HEADER'
```

Имя файла может включать полный путь к файлу, если данные находятся в другом каталоге.

psql\COPY – команда, которая передает данные из клиентской системы, где вы запускаете команду, на сервер базы данных, поэтому файл находится на клиенте. Более высокие привилегии не требуются, поэтому этот метод является предпочтительным.

Если отправлять SQL запрос через соединение другого типа, то можно воспользоваться следующим оператором SQL формы, отметив, что обратная косая черта в начале удалена:

```
COPY sample FROM '/mydatafiledirectory/sample.csv' CSV
HEADER;
```

Здесь оператор COPY использует абсолютный путь для идентификации файлов данных. Этот метод запускается на сервере базы данных и может выполняться только суперпользователем или пользователем, которому прикреплена одна из этих ролей: pg_read_server_files,pg_write_server_files, или pg_execute_server_program. Необходимо убедиться, что серверному процессу

разрешено читать этот файл, затем самостоятельно передавать данные на сервер и, наконец, загружать файл. Эти привилегии обычно не предоставляются, поэтому предпочтительным является ранее показанный метод.

Команда COPY (или\COPY) не создает таблицу, это надо сделать заранее. Отметим также, что команда HEADER ничего не делает, кроме как игнорирует первую строку входного файла, однако имена столбцов из файла .csv не обязательно должны совпадать с именами столбцов таблицы Postgres. Это тоже проблема. Когда указываешь оператор HEADER, а файл не имеет строки заголовка, тогда все, что он делает, это игнорирует первую строку данных. К сожалению, PostgreSQL не может определить, действительно ли первая строка файла является заголовком или нет.

Стандартного инструмента для загрузки данных непосредственно из электронной таблицы в базу данных не существует. Написать макрос электронной таблицы для автоматизации вышеупомянутых задач довольно просто, но здесь создание макросов не рассматривается.

Команда \COPY выполняет инструкцию COPY SQL, поэтому два метода, описанные ранее, очень похожи.

4.6. ЗАГРУЗКА ДАННЫХ ИЗ ТЕКСТОВЫХ ФАЙЛОВ

Загрузка данных в вашу базу данных – одна из самых важных задач. Для стандартной загрузки функция COPY хорошо работает в большинстве случаев, включая загрузку из CSV-файлов, как показано в последнем примере.

Если же нужна расширенная функциональность для загрузки, можно попробовать pgloader, который обычно доступен во всех основных дистрибутивах Postgres.

Для загрузки данных с помощью pgloader необходимо определиться с требованиями и задачами, поэтому разобьем этот процесс на отдельные шаги, как показано ниже:

1. Определитесь с файлами данных и их расположением. Убедитесь, что pgloader указывает на место расположения файлов.

2. Определитесь с таблицей, в которую вы загружаетесь, убедитесь, что у вас есть разрешения на загрузку, и проверьте доступное пространство. Определите тип файла (примеры включают поля фиксированного размера, текст с разделителями и CSV) и проверьте кодировку.

3. Укажите соответствие между столбцами в файле и столбцами загружаемой таблицы. Убедитесь, что вы знаете, какие столбцы в файле не нужны – pgloader позволяет включать только те столбцы, которые необходимы. Определите все столбцы в таблице, в которых у вас нет данных. Необходимо, чтобы они имели значение по умолчанию в таблице, или pgloader должен генерировать значения для этих столбцов с помощью функций или констант?

4. Укажите любые преобразования, которые необходимо выполнить. Наиболее распространенная проблема – это форматы даты, хотя возможны и другие проблемы.

5. Напишите скрипт для pgloader.

6. Скрипт pgloader создаст файл журнала для записи того, была ли загрузка успешной или неудачной, и другой файл для хранения отклоненных строк. Вам нужен каталог с достаточным объемом дискового пространства, если ожидается, что их количество будет большим. Размер каталога примерно пропорционален количеству отклоненных строк.

7. Наконец, подумайте, какие настройки вам нужны для параметров производительности. Это определенно не самое главное во всем процессе загрузки данных из плоских файлов, однако по возможности на это также нужно обратить внимание.

8. Необходимо использовать скрипт для выполнения pgloader. Это не обязательно, но это наилучшее решение, потому что это значительно облегчает решение проблем при загрузке.

Рассмотрим типичный пример из документации pgloader для файла csv.load.

Определите необходимые операции и сохраните их в файле, например csv.load:

```
LOAD CSV
    FROM '/tmp/file.csv' (x, y, a, b, c, d)
    INTO postgresql://postgres@localhost:5432/postgres?csv (a, b, d, c)
    WITH truncate,
        skip header = 1,
        fields optionally enclosed by ' " ',
        fields escaped by double-quote,
```

```
        fields terminated by ','
    SET client_encoding to 'latin1',
        work_mem to '12MB',
        standard_conforming_strings to 'on'
    BEFORE LOAD DO
    $$ drop table if exists csv; $$,
    $$ create table csv (
        a bigint,
        b bigint,
        c char(2),
        d text
        );
$$;
```

Эта команда позволяет нам загрузить следующее содержимое из CSV-файла. Сохраните это в файле, например, file.csv, в каталоге /tmp:

```
Header, with a © sign
 "2.6.190.56","2.6.190.63","33996344","33996351","GB","United Kingdom"
"3.0.0.0","4.17.135.31","50331648","68257567","FR","France"
"4.17.135.32","4.17.135.63","68257568","68257599","CA","Canada"
"4.17.135.64","4.17.142.255","68257600","68259583","US","United States"
"4.17.143.0","4.17.143.15","68259584","68259599","CA","Canada"
"4.17.143.16","4.18.32.71","68259600","68296775","US","United States"
```

Мы можем использовать следующий скрипт для загрузки:

```
pgloader csv.load
```

Вот что загружается в базу данных PostgreSQL:

```
postgres=# select * from csv;
a | b | c | d
-------------+-----------+-----+---------------
33996344 | 33996351 | GB | United Kingdom
50331648 | 68257567 | FR | France
68257568 | 68257599 | CA | Canada
```

68257600 | 68259583 | US | United States
68259584 | 68259599 | CA | Canada
68259600 | 68296775 | US | United States (6 rows)

Как pgloader справляется с ошибками по сравнению с командой COPY. Команда COPY загружает все строки в одной транзакции, поэтому для прерывания загрузки достаточно всего одной ошибки. pgloader разбивает входной файл на фрагменты разумного размера и загружает их по частям. Если некоторые строки в блоке вызывают ошибки, то pgloader будет разбивать его итеративно, пока не загрузит все правильные строки и не пропустит все плохие строки, которые затем сохраняются в отдельном файле rejects для последующей проверки. Такое поведение очень удобно, если у вас есть большие файлы данных с небольшим процентом плохих строк – например, вы можете отредактировать отклоненные строки, исправить их и, наконец, загрузить их с помощью другого запуска pgloader.

Версии из 2.x итераций pgloader были написаны на Python и подключены к PostgreSQL через стандартный клиентский интерфейс Python. Версия 3.x написана на Lips. Да, pgloader менее эффективен, чем загрузка файлов данных с помощью команды COPY, но выполнение команды COPY имеет гораздо больше ограничений: файл должен находиться в нужном месте на сервере, должен быть в нужном формате и не должен вызывать ошибки при загрузке; pgloader имеет дополнительные аппаратные затраты, но он также имеет возможность загружать данные с использованием нескольких параллельных потоков, так что их использование также может быть более быстрым; способность pgloader переформатировать данные с помощью пользовательских функций часто имеет важное значение; использование команды COPY может быть недостаточно.

pgloader также позволяет загружать файлы фиксированной ширины, чего не делает COPY.

Если вам нужно полностью перезагрузить таблицу с нуля, то укажите параметр WITH TRUNCATE в скрипте pgloader.

Существуют также опции для указания SQL, который будет выполняться до и после загрузки данных. Например, можно создать скрипт, который создает пустые таблицы перед загрузкой, можно добавить ограничения, которые выполняются после загрузки или и то, и другое.

4.7. ВНЕСЕНИЕ МАССОВЫХ ИЗМЕНЕНИЙ ДАННЫХ С ИСПОЛЬЗОВАНИЕМ СЕРВЕРНЫХ ПРОЦЕДУР С ТРАНЗАКЦИЯМИ

В некоторых случаях возникает необходимость вносить массовые изменения в свои данные. Во многих случаях необходимо прокручивать данные, внося изменения в соответствии со сложным набором правил.

В этом случае есть несколько вариантов:

– написать один SQL-оператор, который будет делать все;

– открыть курсор, считать строки, а затем внести изменения с помощью клиентской программы;

– написать процедуру, которая использует курсор для чтения строк и внесения изменений с помощью серверного SQL.

Иногда возможно написать один SQL-оператор, который делает все, но если вам нужно сделать больше, чем просто использовать UPDATE, то это очень быстро становится затруднительным. Основная трудность заключается в том, что оператор SQL не подлежит перезапуску, поэтому, если вам нужно будет прервать его, вы потеряете весь свой прогресс.

При этом чтение всех строк через курсор обратно в клиентскую программу может быть очень медленным, поэтому если нужно написать программу такого рода, лучше сделать все это на сервере базы данных.

Создайте таблицу для примера и заполните ее 1000 строк тестовыми данными:

```
CREATE TABLE employee (
empid BIGINT NOT NULL PRIMARY KEY
,job_code TEXT NOT NULL
,salary NUMERIC NOT NULL
);
INSERT INTO employee VALUES (1, 'A1', 50000.00);
INSERT INTO employee VALUES (2, 'B1', 40000.00);
INSERT INTO employee SELECT generate_series(10,1000), 'A2', 10000.00);
```

Теперь напишем процедуру на PL/pgSQL. Процедура похожа на функцию, за исключением того, что она не возвращает никакого значения или объекта. Используем процедуру, потому что она позволяет выполнять несколько транзакций на стороне сервера. Используя процедуры таким образом, можно

разбить задачу на набор небольших транзакций, которые вызывают меньше проблем с раздуванием базы данных и длительностью выполнения транзакций.

В качестве примера рассмотрим случай, когда нам нужно обновить всех сотрудников с уровнем работы A2, дающим каждому человеку 2% надбавки к зарплате:

```
CREATE PROCEDURE annual_pay_rise (percent numeric)
LANGUAGE plpgsql AS $$
DECLARE
c CURSOR FOR
SELECT * FROM employee
        WHERE job_code = 'A2';
BEGIN
FOR r IN c LOOP
UPDATE employee
SET salary = salary * (1 + (percent/100.0))
WHERE empid = r.empid;
IF mod (r.empid, 100) = 0 THEN
COMMIT;
END IF;
END LOOP;
END;
$$;
```

Выполните эту процедуру следующим образом:

```
CALL annual_pay_rise(2);
```

Хорошо бы выпускать регулярные коммиты по ходу работы. Предыдущая процедура закодирована таким образом, что она выдает коммиты примерно через каждые 100 строк. В этом числе нет ничего специального – просто хотим разбить задачу на более мелкие части, будь то количество отсканированных строк или обновленных строк и процедура закодирована таким образом, что она выдает коммиты примерно через каждые 100 строк.

ЗАКЛЮЧЕНИЕ

Изучив данное пособие, внимательный студент познакомился с основными идеями и принципами построения систем управления базами данных и получил представление о том, как эти идеи развивались и изменялись на протяжении десятилетий. Мы показали, как идеи и принципы влияют на модели и языки, а модели и языки реализуются в промышленных СУБД.

Не переполняя текст деталями, мы рассказали, как организовывать взаимодействие различных моделей и языков, необходимых для реализации прикладных систем, для того чтобы эти системы получались высококачественными, надежными и эффективными.

Мы показали, как рекомендации теории реализуются в системе управления базами данных PostgreSQL. За три десятилетия своего развития эта система из небольшого экспериментального прототипа объектно-ориентированной СУБД превратилась в мощную систему, которая поддерживает разнообразные модели данных и методы разработки приложений, включает высокоэффективные средства выполнения запросов и развитые средства управления конкурентным доступом. Современные версии PostgreSQL предоставляют все необходимое для создания высокопроизводительных и высоконадежных информационных систем.

Благодаря многообразным средствам расширения система PostgreSQL открыта для добавления новой функциональности, а возможность добавления новых источников данных, не обязательно хранящихся под управлением PostgreSQL, позволяет создавать неоднородные распределенные системы.

В настоящее время система PostgreSQL является наиболее мощной среди систем с открытым кодом и успешно конкурирует с коммерческими системами. Технологии баз данных постоянно развиваются.

Это связано как с появлением новых областей применения и новых средств разработки приложений, так и с расширением возможностей вычислительных систем, на которых работают серверы баз данных.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. **Гарсиа-Молина, Г.** Системы баз данных. Полный курс / Г. Гарсиа-Молина, Д. Д. Ульман, Д. Уидом : пер. с англ. – М. : Вильямс, 2003. – 1088 с.
2. **Грофф, Дж. Р.** SQL. Полное руководство / Дж. Р. Грофф, П. Н. Вайнберг, Э. Оппель Дж. : пер. с англ. – 3-е изд. – М. : Вильямс, 2015. – 960 с.
3. **Дейт, К.** Дж. Введение в системы баз данных / К. Дж. Дейт : пер. с англ. – 8-е изд. – М. : Вильямс, 2005. – 1328 с.
4. **Новиков, Б.** Настройка приложений баз данных / Б. Новиков, Г. Домбровская. – СПб. : БХВПетербург, 2012. – 240 с.
5. **Новиков, Б.** Основы технологий баз данных / Б. Новиков, Е. Горшкова, Н. Графеева. – М. : Postgres Professional, 2018. – 182 с.
6. **Селко, Д.** Стиль программирования Джо Селко на SQL / Д. Селко : пер. с англ. – М. : Русская редакция ; СПб. : Питер, 2006. – 206 с.
7. **Официальный сайт** PostgreSQL. – URL : http://www.postgresql.org
8. **Postgres** Professional http://postgrespro.ru
9. Учебные курсы. – URL : Postgres Professional http://postgrespro.ru/education/courses
10. **Рогов, Е.** Индексы в Postgre SQL / Е. Рогов. – 2017. – URL : https://habr.com/company/ postgrespro/blog/326096/
11. **Селко, Д.** SQL для профессионалов. Программирование / Д. Селко. – М. : Лори, 2004. – 456 с.
12. **Лекции** лауреатов премии Тьюринга за первые 20 лет / под ред. Ю. Баяковского. – М. : Наука, 1993. – 560 с.
13. **Черкасова, П.** Бесшовное проектирование: снова о несоответствии импеданса / П. Черкасова, Б. Новиков // Программирование. – 2006. – Т. 32, № 5. – С. 268– 275.

ОГЛАВЛЕНИЕ

СПИСОК СОКРАЩЕНИЙ ………………………………………………….. 3
ВВЕДЕНИЕ ……………………………………………………………… 4
1. ПЕРВЫЕ ШАГИ ……………………………………………………… 5
1.1. Знакомство с PostgreSQL14 …………………………………….. 5
1.2. Скачивание и установка PostgreSQL ………………………………… 11
1.3. Подключение к серверу PostgreSQL ………………………………… 12
1.4. Включение доступа для сети/удаленных пользователей ……………… 15
1.5. Использование инструмента pgAdmin4GUI ………………………… 18
2. ИЗУЧЕНИЕ БАЗ ДАННЫХ ……………………………………………… 23
2.1. Тип СУБД PostgreSQL и его версии ………………………………… 23
2.2. Время безотказной работы …………………………………………… 26
2.3. Расположение файлов сервера базы данных ……………………….. 27
2.4. Список баз данных на сервере базы данных ……………………….. 32
2.5. Таблицы в базе данных ……………………………………………… 34
3. КОНФИГУРАЦИЯ СЕРВЕРА ………………………………………… 39
3.1. Планирование новой базы данных ………………………………... 39
3.2. Установка параметров конфигурации для сервера базы данных ……. 40
4. ТАБЛИЦЫ И ДАННЫЕ ……………………………………………... 42
4.1. Выбор подходящих имен для объектов базы данных ………………… 43
4.2. Выявление и удаление дубликатов ………………………………….. 48
4.3. Поиск уникального ключа для набора данных …………………….. 58
4.4. Генерация тестовых данных ………………………………………… 62
4.5. Загрузка данных из электронной таблицы …………………………. 68
4.6. Загрузка данных из текстовых файлов ……………………………… 71
4.7. Внесение массовых изменений данных с использованием серверных процедур с транзакциями 75
ЗАКЛЮЧЕНИЕ ………………………………………………………… 77
СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ ……………………………… 78

Учебное электронное издание

БУРЦЕВА Елена Васильевна
РАК Игорь Петрович
ПЛАТЕНКИН Алексей Владимирович

БАЗЫ ДАННЫХ

Учебное пособие

Редактирование Е. С. Мордасовой
Графический и мультимедийный дизайнер Т. Ю. Зотова
Обложка, упаковка, тиражирование Е. С. Мордасовой

ISBN 978-5-8265-2650-7

Подписано к использованию 05.10.2023.
Тираж 50 шт. Заказ № 116

Издательский центр ФГБОУ ВО «ТГТУ»
392000, г. Тамбов, ул. Советская, д. 106, к. 14
Телефон: (4752) 63-81-08
E-mail: izdatelstvo@tstu.ru